SESSION

FAULT-TOLERANT SYSTEMS + FAULT DETECTION METHODS + FAULT MANAGEMENT AND TOOLS

Chair(s)

TBA

A Performance Comparison of Resource Allocation Policies in Distributed Computing Environments with Random Failures

Bhavesh Khemka[1], **Anthony A. Maciejewski**[1], **and Howard Jay Siegel**[1,2]
[1]Department of Electrical and Computer Engineering, [2]Department of Computer Science,
Colorado State University, Fort Collins, Colorado, USA

Abstract— *The problem of efficiently assigning tasks to machines in heterogeneous computing environments with uncertainty in the availability of the compute resources is a challenging one. Previous research has looked at designing heuristics to maximize the total reward earned by completing tasks in an environment where compute nodes may randomly fail. The rewards associated with the tasks are earned if they are successfully executed before their deadlines. The goal of the resource allocation policies is to maximize the cumulative reward earned. We use heuristics from the literature and improved versions of some of the heuristics to perform the resource allocation decisions. We conduct extensive experiments to compare the performance of these heuristics in a variety of simulation environments. The goal of the study is to be able to recommend a heuristic to use based on the system environment. Our experiments show that different heuristics perform the best in different environments.*

Keywords: mapping, resource allocation, fault-tolerant, heterogeneity, rewards, deadlines

1. Introduction

Distributed computing is currently used to solve a host of problems where different tasks are mapped to separate machines for execution. These environments may be heterogeneous, which means different tasks may have varied execution times on the different machines. This makes it difficult to assign tasks to machines to optimize for a given performance metric. The process of allocating tasks to machines for execution is commonly referred to in the literature as "resource allocation" or "mapping." The mapping and scheduling problem has been known to be NP-Complete [1], and therefore one must use heuristics to get a solution to this problem. It is common for failures to randomly occur in the compute resources of these large-scale distributed systems. As a result, it becomes even more difficult to make resource allocation decisions while being aware of such failures. In this study, the goal of our resource allocation procedures is to maximize the total reward earned while executing the tasks in an environment where the machines may randomly fail.

This work was supported by the National Science Foundation under grant number CNS-0905399, and by the Colorado State University George T. Abell Endowment.

Fault-tolerance in distributed computing environments has been extensively studied. Check-pointing tasks to avoid restarting them from the beginning, in case they fail, is a common method used to alleviate the damage caused by the failure of resources [2], [3], [4], [5]. Replicating tasks is another method used to improve the reliability of the system [6], [4], [7], [8].

Shestak et al. [9] addressed the problem of making resource allocation decisions while being aware of the mean fail rates of the compute resources. To model a harsh environment the machines were made to have high probabilities of failure. They assumed that a bag of tasks is available and ready for execution. Each of those tasks has a reward and a deadline associated with it. The reward of a task is earned if the task is successfully completed by its deadline. If a machine fails while executing the task, the task is returned to the batch and may be re-mapped to another machine for execution. The goal of the resource allocation heuristics is to maximize the total reward that can be earned by executing the tasks before their respective deadlines expire.

The contributions of this paper are: (a) an enhancement in the prediction mechanism for some of the heuristics from the literature [9], and (b) a study of the performance of different heuristics on various environments that differ in their types of heterogeneities, rates for task executions and machine failures, and task re-mapping policies. We compare the relative performance of the heuristics in each of these simulated environments, and recommend the heuristic that one should choose to obtain the highest reward for each environment.

2. Problem Statement

The environment we model in this study and the goal of our resource allocation procedures are based on the work of Shestak et al. [9]. Each task has an associated reward that is earned if its computation is successfully completed by its deadline. The compute resources are assumed to have high probabilities of failure. When a machine failure occurs while executing a task, the task is returned to the batch and is eligible to be mapped to another machine. The task may fail multiple times, and it can be continually remapped to

constant	mach A	mach B	mach C
task class 1	4	4	4
task class 2	4	4	4
task class 3	4	4	4

column-varying	mach A	mach B	mach C
task class 1	5	2	4
task class 2	5	2	4
task class 3	5	2	4

inconsistent	mach A	mach B	mach C
task class 1	6	5	7
task class 2	9	3	4
task class 3	2	10	1

task-mach-consistent	mach A	mach B	mach C
task class 1	1	2	7
task class 2	3	4	9
task class 3	5	6	10

Fig. 1: Sample Estimated Time to Compute (ETC) matrices modelling different types of heterogeneity in an environment with three task classes and three machines

machines as long as its deadline has not expired. The goal of the resource allocation policies will be to map tasks to machines to maximize the total reward earned from all the tasks. In this study, we say that a heuristic is "robust" [10] if it can earn reward in an environment with uncertainties in task execution times and machine failure times. The higher the reward a heuristic earns, the more robust it is to such uncertainties.

We work with a bag of tasks that was available at the start of the simulation. Whether or not failed tasks are allowed to come back for mapping events may be a policy decision. Therefore, we consider both cases (allowing and not allowing failed tasks to re-map), and simulate both environments for experimentation purposes. We also model various types of heterogeneity of the computing environment, and gauge the performance of the various resource allocation procedures.

3. Environment Modeling

3.1 Modeling the Task Execution Times

An Estimated Time to Compute (ETC) matrix is used to model the execution time characteristics of the various tasks in the heterogeneous system. We use task classes to group together tasks that have similar execution time characteristics. Each entry t_{ij} in the ETC matrix gives the mean execution time of tasks of class i on machine j of our heterogeneous suite. The actual execution time of the tasks are modeled using exponential distributions with the means obtained from the entries of the ETC matrix. We use exponential distributions to model task completion times, based on the tests conducted at Ricoh InfoPrint [11]. For simulation purposes, we create and use synthetic workloads, but in real-world environments one could build such a matrix based on historical data.

We model and gauge the performance of various resource allocation policies under different types of ETC matrices. The different ETC matrices are used to model various types of heterogeneity of computing systems. We model four types of ETC matrices. Sample 3 x 3 ETC matrices for each of these types are shown in Figure 1. We model homogeneous workloads and homogeneous compute resources by having a fixed value for all the entries in the ETC matrix. We call this type of ETC matrix *constant*. We model another environment wherein the workload can be considered homogeneous, but the compute resources can be considered to have different computational capabilities. We model such an environment by having unique values for each of the columns of the ETC matrix, and refer to this matrix as *column-varying*. A completely heterogeneous environment is modeled by having random values for each cell in the ETC matrix. In an environment, modeled by such a matrix, it is possible (and likely) for a machine to be better than another machine for a particular task class and worse for another. Such a matrix is referred to as *inconsistent*. If we independently sort the elements within each of the rows of such an *inconsistent* ETC, and then independently sort the entries in each of the columns, we obtain what we call a *task-mach-consistent* matrix. In such an environment, if a machine executes a task faster than another machine, then it will do so for all tasks. Similarly, if a task executes faster than another task on a single machine, then it will do so on all machines. This type of matrix is well suited for some of the heuristics discussed later.

3.2 Modeling the Machine Failures

It has been shown in the literature that exponential distributions can be used to stochastically model hardware failures [12], [13]. In our environment, each machine has an exponential distribution associated with it to model the probability of failure, and for each machine j, we have a failure rate represented by λ_j.

Shestak et al. [9] have shown how a distribution of processor availability can be obtained using the average execution time of tasks on a machine j (referred to as t_j^{av}), and the failure rate of the machine λ_j. This probability mass function consists of as many pulses as there are machines in the environment. The distribution gives the relative probability of each machine to become available

for a mapping event. Machines become available for a mapping event under two scenarios: if they have successfully completed a task, or if they have encountered a failure. The distribution sorts the machines in an ascending order of their "quality." A machine has better "quality" if it has a lower value for $\lambda_j t_j^{av}$. Therefore, a machine that has a lower failure rate and/or a lower value for mean execution time, will be considered better. The Cumulative Distribution Function (CDF) of this probability distribution will be used to guide resource allocation decisions by some heuristics.

4. Resource Allocation Policies

4.1 Overview

In our environment, a mapping decision (deciding which machine to map a task to) is made whenever a machine becomes available for executing a task. Machines become available in two cases: when they successfully complete a task that was assigned to them, or when they have encountered a failure. We make an assumption that failed machines get instantly repaired and are available for executing tasks immediately. We use heuristics from the literature [9] as well as modifications to these heuristics to make mapping decisions.

4.2 Heuristics from the Literature

4.2.1 Reward Heuristics:

There are two heuristics that directly try to optimize the reward [9]. They are the *Reward* heuristic and *Expected Reward* heuristic. In *Reward*, whenever a mapping event occurs, the task that has the highest value for reward is assigned to the machine that just became available. In *Expected Reward*, when a machine becomes available for a mapping event, the task with the highest value for expected reward is assigned to it.

For a task i, $\underline{P_i(t)}$ is the probability (computed at time t) of successfully completing task i through multiple assignments before its deadline expires. The expected reward for a task i is given by the product $r_i P_i(t)$, where r_i is the reward that this task can earn if completed successfully. For any machine j, we represent by $\underline{p_j}$ the probability of machine j being available for an assignment. Also, the term $\underline{V_i(t)}$ is the estimated number of reassignments that task i may undergo starting at time t up to its deadline. The derivations of these terms are shown in [9]. $P_i(t)$ is calculated using the equation shown below.

$$P_i(t) = 1 - \left(\sum_{j=1}^{M} p_j (1 - e^{-\lambda_j t_{ij}}) \right)^{V_i(t)} \quad (1)$$

The term $(1 - e^{-\lambda_j t_{ij}})$ gives the probability of task i failing on machine j. This factor is weighed with the probability of machine j being available for an assignment (p_j), and therefore the weighted sum, $\sum_{j=1}^{M} p_j (1 - e^{-\lambda_j t_{ij}})$, gives the probability of failure when task i is mapped to a machine. Therefore, Equation 1 represents the probability that task i will successfully complete before its deadline, even through multiple assignments.

4.2.2 Matching Heuristics:

There are two heuristics in [9] that use the concepts of the Derman-Lieberman-Ross (DLR) Theorem [14] to guide mapping decisions. We call them the *Matching* heuristic and *Expected Matching* heuristic. A brief overview of the DLR theorem is given below, followed by the *Matching* heuristics that are implemented using the DLR concept.

The DLR theorem [14] provides an algorithm for optimally assigning a set of available workers to incoming jobs. Each incoming job is assumed to have a reward value associated with it. Each worker is assumed to have a probability (that represents the quality and skill of the worker), with which the reward earned for a job is scaled. It is also assumed that one has the distribution from which the reward values for all the incoming tasks are sampled from. Using the distribution of the reward values of the incoming tasks, and the notion of the skill of the workers (determined by their probabilities), the DLR method describes an algorithm that maps high reward tasks to better skilled workers and low reward tasks to lesser skilled workers. The distribution that dictates the reward values of the incoming jobs is vital to making these decisions.

The *Matching* heuristics [9] try to implement the DLR concept within the resource allocation problem. In our environment, machines become available for mapping events, and a task needs to be assigned to them. This is analogous to jobs coming in and looking for a worker that can be assigned to them. The distribution described in Section 3.2 is used to describe the quality and the likelihood of the incoming machine, analogous to the distribution that governs the likelihood of various reward values for the incoming job. The only other factor that needs to be accounted for is the ranking of the tasks, analogous to the ranking of the workers. It is in this aspect that the *Matching* and the *Expected Matching* heuristics differ. In *Matching*, the tasks are sorted based on their reward values. In *Expected Matching* the tasks are sorted based on their value of expected reward. As before, expected reward of a task i is given by the product $r_i P_i(t)$.

4.3 Modifications to Heuristics

We modify Equation 1 to incorporate the knowledge of the machine that just became available for a mapping event. Let us call the machine that just became available to be machine J. The probability that this machine will become available p_J will be 1, and by a similar logic $p_j = 0, \forall j \neq J$. Therefore, the summation term for this mapping event will reduce to $(1 - e^{-\lambda_J t_{iJ}})$. We know that this counts as an assignment for task i, and therefore we extract the term

$(1 - e^{-\lambda_J t_{iJ}})$ out, and reduce the count of the number of reassignments of task i (denoted by $V_i(t)$) by one. This gives us our new equation for $P_i(t)$.

$$P_i(t) = 1 - (1 - e^{-\lambda_J t_{iJ}}) \times \left(\sum_{j=1}^{M} p_j (1 - e^{-\lambda_j t_{ij}}) \right)^{V_i(t)-1} \quad (2)$$

We create the *Latest Expected Reward* and the *Latest Expected Macthing* heuristics that are similar to the *Expected Reward* and *Expected Matching* heuristics, but with the difference that they use Equation 2 for their expression of $P_i(t)$ instead of Equation 1. This is to denote the fact that they use the latest information to compute $P_i(t)$.

For experimentation purposes, we also model an environment where tasks are not allowed to come back when the machine they were assigned to failed. In such an environment, the value of $P_i(t)$ is simply calculated using the equation given below.

$$P_i(t) = e^{-\lambda_J t_{iJ}} \quad (3)$$

5. Simulation Setup

In this study, the workload that we are modeling consists of independent tasks, i.e., no communication is required between the individual tasks, and there are no precedence constraints. Each task is sequential (i.e., not decomposable into parallel parts). It is also assumed that each machine can handle only one task at a time (no multitasking).

We modeled different ETC matrices, as described in Section 3.1. The entries of the ETC matrix represent the mean execution time values of the different task classes on the different machines. We modeled two types of environments; the *narrow* and the *broad* environments. For the *narrow* environment, the entries of the ETC matrix were uniformly picked at random from the range [0.5, 4.0], whereas for the *broad* environment they were uniformly picked at random from the range [0.5, 9.5].

The values of the mean times to failure for the machines were also randomly picked from a range depending on the environment that was being modeled. For the *narrow* environment, the range was [0.6, 1.0], whereas, for the *broad* environment, the range was [0.6, 2.375]. Therefore, the *broad* environment has wider ranges for the possible values of the execution times of the tasks and the times between failure for the machines in comparison to the *narrow* environment.

Similar to the environment studied by Shestak et al. [9], we model an environment with 200 tasks, six machines, and five task classes. Each task is randomly assigned to one of the five task classes. The ETC matrix has task classes along its rows, and machines along its columns, and therefore the size of the ETC matrix is 5 x 6. For each task, the reward value was assigned by randomly picking an integer in the range [1, 100]. All the tasks in a task class have a common deadline. The deadline for a task class was set to six times the longest execution time of this task class across the machines.

There were three main parameters that we varied to alter the environment being modeled. The first was whether or not we allow failed tasks to return back to the batch for further remapping. The second was whether we use the *narrow* or *broad* environment. The final parameter was the type of ETC matrix used. We modeled four types of ETC matrices: *constant*, *column-varying*, *inconsistent*, and *task-mach-consistent*. We ran tests with all combinations of these various parameters.

6. Experimental Results and Analysis

The goal of this study is to compare and evaluate the performance of the heuristics under a variety of scenarios to be able to choose which heuristic to use for different environments. Therefore, it serves to only compare the relative performance of the heuristics with each other under the various scenarios, as opposed to comparing the absolute performance of a heuristic across the scenarios. It would also be inaccurate to make such a comparison, because the different scenarios have different execution time characteristics and different handling methods for dropped tasks. These can result in different values for the total reward earned by the same heuristic under these varied environments.

For each simulation case, 100 trials were performed and the results were averaged and 95% confidence intervals were calculated. For each trial, new values were used for the following: the entries of the ETC matrix, reward values of the tasks, and fail rates of the machines.

We observed that the *Expected Reward* heuristic did not perform better than the *Latest Expected Reward* heuristic for any of the test cases. The *Latest Expected Reward* heuristic is able to use the most recent information to its advantage while calculating the probability of a task successfully completing. The significant benefit earned by using Equation 2 as opposed to Equation 1 comes from the fact that we are able to factor out the current machine from the summation, thus avoiding distorting the probability of failure for this first mapping event.

From our experiments, we also observed that the *Latest Expected Matching* heuristic performed better than both the *Matching* and the *Expected Matching* heuristics. Therefore, in Figure 2 we only show the results for the *Reward*, *Latest Expected Reward*, and *Latest Expected Matching* heuristics. Each of these three heuristics perform better than the other two heuristics for at least some of the environments.

Figure 2 shows the results for the *Reward*, *Latest Expected Reward*, and *Latest Expected Matching* heuristics for the various types of environments that we modeled. Figures 2(a) and 2(b) show results from the case where re-mapping of failed tasks is allowed. Figures 2(c) and 2(d) shows

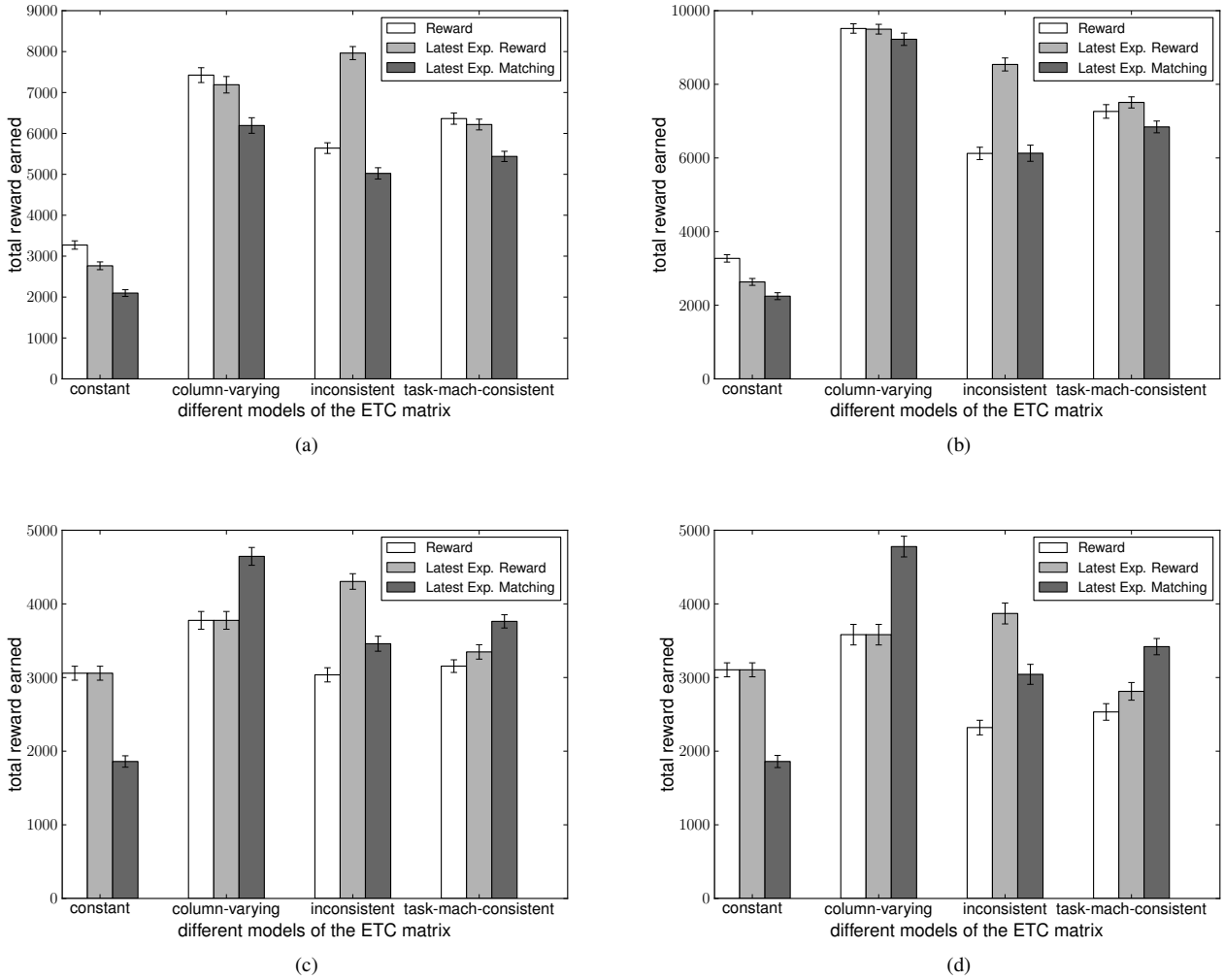


Fig. 2: Results showing the total reward earned by the *Reward*, *Latest Expected Reward*, and the *Latest Expected Matching* heuristics in (a) the *narrow* environment with re-mapping allowed, (b) the *broad* environment with re-mapping allowed, (c) the *narrow* environment with no remapping, and (d) the *broad* environment with no remapping. The results are averaged over 100 trials and the error bars show 95% confidence intervals.

results with the case where each task can be mapped to a machine only once. Figures 2(a) and 2(c) have results from the *narrow* environment. This determines the ranges of the mean execution times of the tasks and the mean fail times of the machines. Figures 2(b) and 2(d) show results for the case with the *broad* environment. For each of these charts, along the independent axis are the results of the three heuristics for the various types of ETC matrices that we modeled, i.e., *constant*, *column-varying*, *inconsistent*, and *task-mach-consistent*. It is worth noting that our goal will be to compare the relative performance of the three heuristics for any given scenario (i.e., comparing the bars within the set of three) as opposed to comparing the results across scenarios.

The *Reward* heuristic performs the best among the three heuristics when we have a completely homogeneous environment (i.e., modeled by the *constant* ETC matrix) irrespective of whether or not tasks are allowed to come back, and whether we use the *narrow* or the *broad* environment. Note that in the case when tasks are not allowed to come back (Figures 2(c) and 2(d)) and when the *constant* or the *column-varying* ETC matrices are used, the *Latest Expected Reward* and the *Reward* heuristic have equal results. On average, the total reward earned by the *Reward* heuristic was at least a 45% improvement over that earned by the *Latest Expected Matching* heuristic when the *constant* ETC matrix was used. This can be attributed to the fact that the *Reward* heuristic directly optimizes for reward, and in an environment where all tasks perform similarly across all machines, the benefits

obtained by performing a matching of tasks and machines is reduced.

In cases where the environment is heterogeneous (i.e., models of the ETC matrix other than *constant*), the *Reward* heuristic beats the *Latest Expect Matching* heuristic when failed tasks are allowed to come back for mapping events (Figures 2(a) and 2(b)). This trend is reversed when failed tasks are not allowed to come back (Figures 2(c) and 2(d)). The *Latest Expected Matching* always does better than the *Reward* heuristic (except the case with the *constant* ETC) when failed tasks are not allowed to come back for re-mapping. Also, in general the relative performance of the *Latest Expected Matching* heuristic is better in such environments. Unlike the other heuristics, the *Latest Expected Matching* heuristic, tries to match "better" tasks (those that have a higher value for expected reward) to "better" machines (those that execute faster, and fail less often). This is particularly helpful when re-mapping is not allowed, because in this case, each task only gets one chance to be mapped to a machine. The *Latest Expected Matching* heuristic performs the best in this case because it holds on to the better tasks until a good machine comes in for a mapping event. The *Reward* (or the *Latest Expected Reward*) heuristic simply assign the task with the highest reward (or expected reward) to the next available machine.

The *Latest Expected Matching* heuristic performs better than the *Latest Expected Reward* heuristic when the environment uses the *column-varying* ETC matrix or the *task-mach-consistent* ETC matrix and when failed tasks are not allowed to re-map. This is because in such environments it becomes easier for the *Latest Expected Matching* heuristic to be able to designate some machines as being better than the others. This helps the heuristic rank the machines in terms of their "quality" and as a result provides better mapping decisions. Moreover, when we use the *broad* environment as opposed to the *narrow* environment, the relative performance benefit of this heuristic increases. This is because there is more variance in the performance of the machines and the *Latest Expected Matching* heuristic is able to use that to its advantage as it makes its decisions by ranking machines in terms of their goodness. The more the difference between the performance of the "good" and "bad" quality machines, the more the benefit of using the *Latest Expected Matching* heuristic.

When we use an *inconsistent* ETC matrix it becomes hard for the *Latest Expected Matching* heuristic to rank the machines. This is because in a highly inconsistent heterogeneous environment, a machine may perform better than another machine for one task, but may perform worse for another task. This makes it harder for the *Latest Expected Matching* algorithm to be able to rank the machines in a global manner in terms of their "quality." Therefore, we see that the *Latest Expected Reward* heuristic always performs better when we model the environment with an *inconsistent* matrix. It performs much better than the *Reward* heuristic in all *inconsistent* ETC matrix cases, because it calculates the expected reward (viz. the probability of earning some reward amount) by looking at the execution time values of the tasks. The *Reward* heuristic fails to look at the heterogeneity of the system while making its mapping decisions, and therefore performs poorly.

In summary, the *Latest Expected Matching* heuristic performs the best when the environment being modeled is similar to the environment of the DLR theorem [14] (on which this heuristic is based), i.e., re-mapping of tasks is not allowed, and the machines can be ranked clearly in terms of their performance (modeled by *column-varying* and *task-mach-consistent* types of ETC matrices). The *Reward* heuristic performs the best when the environment is completely homogeneous (modeled by the *constant* type of ETC matrix), because the execution times are the same for all tasks and machines. The *Latest Expected Reward* heuristic performs the best compared to the other heuristics in a highly heterogeneous environment (modeled by the *inconsistent* type of ETC matrix) because it optimizes for not just the reward value but also the likelihood of earning that reward.

7. Related Work

The scheduling problem has been widely studied in heterogeneous computing environments (eg., [15], [16], [17]). It is important to make the resource allocations be fault tolerant, especially in distributed and grid computing environments. Various techniques have been used to cope with the ill-effects of failures of compute resources. Checkpointing and rollback-recovery are common techniques used to avoid having to restart failed tasks from the beginning (e.g., [2], [3], [4], [5]) (as mentioned in Section 1). Another method used to improve the reliability of the system, in terms of increasing the chances of completing tasks, is to run replicas of the tasks on multiple compute resources (e.g., [6], [4], [7], [8]).

Shestak et al. [9] addressed the problem of maximizing the reward earned by the tasks in an environment where the compute nodes may randomly fail. Their work used the concepts of a theorem introduced by Derman et al. [14]. There have been other works on scheduling that look at maximizing reward earned by the tasks [18], but they do not model environments where the machines tend to fail. Our study builds on the work done in [9] to perform a comparative study of the performance of the heuristics under a variety of system environments.

8. Conclusions and Future Work

The goal of this study was to be able to model and characterize various system environments and gauge the relative performance of fault-tolerant heuristics in these environments. This study extends the work of Shestak et

al. [9] by addressing a similar problem, but performing extensive tests on a wide-range of environments. We also modified and improved the prediction mechanism of the *Expected Reward* and the *Expected Matching* heuristics by using the latest information we have about the system. We simulated a variety of environments by changing different attributes associated with the environment. Our results show that the *Reward*, *Latest Expected Reward*, and the *Latest Expected Matching* heuristics have different strengths and weaknesses, therefore performing better or worse depending on the environment.

One direction for future work is introducing a delay before a failed machine returns for a mapping event. This will help us to more closely model a realistic environment. Also, we may try to modify the *Latest Expected Matching* heuristic to make it more heterogeneity-aware. It would also be interesting to modify the workload to have tasks whose reward values degrade with time, instead of having a fixed reward value until a hard deadline.

Acknowledgments: *The authors thank Ryan Friese and Mark Oxley for their valuable comments.*

References

[1] M. R. Gary and D. S. Johnson, *Computers and Intractability: A guide to the theory of NP-Completeness*. W. H. Freeman and Co., 1979.

[2] D. Manivannan, "Checkpointing and rollback recovery in distributed systems: existing solutions, open issues and proposed solutions," in *Proceedings of the 12th International Conference on Systems*. World Scientific and Engineering Academy and Society (WSEAS), 2008, pp. 569–574.

[3] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23–31, Jan. 1987.

[4] Y. Zhang, A. Mandal, C. Koelbel, and K. Cooper, "Combined fault tolerance and scheduling techniques for workflow applications on computational grids," in *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09)*, 2009, pp. 244–251.

[5] B. Nazir, K. Qureshi, and P. Manuel, "Adaptive checkpointing strategy to tolerate faults in economy based grid," *The Journal of Supercomputing*, vol. 50, pp. 1–18, 2009.

[6] A. Litke, D. Skoutas, K. Tserpes, and T. Varvarigou, "Efficient task replication and management for adaptive fault tolerance in mobile grid environments," *Future Generation Computer Systems*, vol. 23, no. 2, pp. 163–178, Feb. 2007.

[7] Y. Oh and S. Son, "Scheduling real-time tasks for dependability," *The Journal of the Operational Research Society*, vol. 48, no. 6, pp. 629–639, June 1997.

[8] Q. Zheng, B. Veeravalli, and C.-K. Tham, "On the design of fault-tolerant scheduling strategies using primary-backup approach for computational grids with low replication costs," *IEEE Transactions on Computers*, vol. 58, pp. 380–393, Mar. 2009.

[9] V. Shestak, E. K. P. Chong, A. A. Maciejewski, and H. J. Siegel, "Probabilistic resource allocation in heterogeneous distributed systems with random failures," *Journal of Parallel and Distributed Computing*, accepted to appear.

[10] S. Ali, A. A. Maciejewski, H. J. Siegel, and J.-K. Kim, "Measuring the robustness of a resource allocation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 7, pp. 630–641, jul 2004.

[11] InfoPrint. (accessed Mar. 2012). [Online]. Available: http://www.infoprint.com/internet/ipww.nsf/vwWebPublished/print-infoprint-5000-en

[12] C. E. Ebeling, *Introduction to Reliability and Maintainability Engineering*. Waveland Pr Inc, 2005.

[13] M. Rausand and A. Høyland, *System Reliability Theory: Models, Statistical Methods, and Applications*. Wiley-Interscience, 2008.

[14] C. Derman, G. J. Lieberman, and S. M. Ross, "A sequential stochastic assignment problems," *Management Science*, vol. 18, no. 7, pp. 349–355, Mar. 1972.

[15] T. D. Braun, H. J. Siegel, N. Beck, L. Boloni, R. F. Freund, D. Hensgen, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810Ű–837, June 2001.

[16] V. Shestak, J. Smith, H. J. Siegel, and A. A. Maciejewski, "Stochastic robustness metric and its use for static resource allocations," *Journal of Parallel and Distributed Computing*, vol. 68, no. 8, pp. 1157–1173, Aug. 2008.

[17] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 107Ű–121, Nov. 1999.

[18] L. D. Briceno, B. Khemka, H. J. Siegel, A. A. Maciejewski, C. Groer, G. Koenig, G. Okonski, and S. Poole, "Time utility functions for modeling and evaluating resource allocations in a heterogeneous computing systems," in *20th Heterogeneity in Computing Workshop (HCW 2011), in the procedings of the 25th International Parallel and Distributed Processing Symposium (IPDPS 2011)*, May 2011, pp. 7–19.

GPU Acceleration of Genetic Algorithms for Subset Selection for Partial Fault Tolerance

D. Foster
Electrical and Computer Engineering Department, Kettering University, Flint, MI, USA

Abstract - *As reconfigurable logic devices see increasing use in aerospace and terrestrial applications, fault tolerant techniques are being developed to counter rising susceptibility due to decreasing feature sizes. Applying fault-tolerance to an entire circuit induces unacceptable area and time penalties, thus some techniques trade area for fault tolerance. Area-Constrained Partial Fault Tolerance (ACPFT) is a methodology that explicitly accepts a device's resources as an input and attempts to find a maximally fault-tolerant subset, but determining an optimal partition is still an open problem. While ACPFT originally used heuristics for subset selection, a modification called ACPFT-GA has been developed that uses genetic evolution to provide significantly better fault coverage in many applications. However, its running time is substantially longer than standard ACPFT and may be prohibitive. This paper presents a GPU-accelerated version of ACPFT-GA that has executed over 27 times faster than CPU versions, allowing ACPFT-GA to better scale to larger circuits.*

Keywords: Genetic algorithms, partial fault tolerance, reconfigurable logic, GPU programming

1 Introduction

Two options for a system's processing device are general purpose processors (GPU) and application specific integrated circuits (ASIC). The GPU offers great flexibility but low relative computational power. An ASIC can be designed to provide the greatest processing capability, however this requires a lengthy and very expensive design process, and it is extremely costly for small production runs. A reconfigurable logic device such as a field-programmable gate array (FPGA) offers an attractive third alternative. They provide high levels of computational power like ASICs, yet their ability to be reprogrammed gives flexibility like GPUs. They are off-the-shelf devices and therefore do not have the lead times of ASICs. These features make them common choices in the low production runs of aerospace applications, and they are increasingly used in terrestrial systems. However, they may contain millions of bits to store configurations, and this makes them more susceptible to faults caused by electromagnetic radiation than GPUs and ASICs. Aerospace systems are currently concerned with errors due to single event upsets (SEUs), and as transistor feature sizes continue to shrink, terrestrial systems are also becoming wary[1, 2].

Many applications using reconfigurable logic are not safety critical. A failure can be tolerated by ignoring the error and continuing, such as in video playback. In other cases, the operation can be reattempted, such as retransmitting dropped packets in network communication. However, a reduction of faults would clearly improve the user experience. Since systems implemented with reconfigurable logic invariably leave a portion of the device unused, these extra device resources can be leveraged to provide some level of partial-fault tolerance and reduce the fault rate. The problem of applying partial fault-tolerance can then be formulated as follows. The logic cells contained with the original circuit must be partitioned into a protected subset and non-protected subset such that the fault-coverage is maximized given a set amount of additional logic resources. With current logic devices having hundreds of thousands of logic cells, this presents a gigantic solutions space.

A method of partial fault tolerance called Area-Constrained Partial Fault Tolerance (ACPFT) has been developed that accepts a circuit's available area as an input and finds a maximally fault-tolerance version of the circuit. This initial implementation utilizes difference heuristics to determine a partition, and it generally executes very quickly. A second version called ACPFT – Genetic Algorithm (ACPFT-GA) uses genetic evolution to explore the solution space. It was found to produce significantly more fault-tolerant circuits in expected application spaces, but the running times of ACPFT-GA can be two orders of magnitude larger than ACPFT's. To provide the fault coverage of ACPFT-GA with a more acceptable execution time, the research presented here accelerates ACPFT-GA using NVIDIA CUDA, which is a popular programming extension for running scientific computations directly on massively-parallel graphics processors. This results in an average speed-up of around 17 to 18 times over standard ACPFT-GA with some cases showing speedups of over 27 times.

This paper is organized as follows. Section 2 reviews the key concepts of ACPFT-GA, other efforts to accelerate genetic algorithms with CUDA, and the key considerations when developing with CUDA. Section 3 describes the implementation of ACPFT-GA, the tool chain, and the particulars of casting it as a CUDA-based algorithm. Section 4 presents the experimental results, and Section 5 concludes the paper.

2 Background

2.1 Partial Fault Tolerance

Since this paper focuses on the acceleration of ACPFT-GA and not the introduction of ACPFT-GA itself, readers are referred to [3] and [4] for the justifications of partial fault tolerance in reconfigurable logic. These summarize the advances in partial fault tolerance along with alternatives to ACPFT such as the BYU-LANL partial TMR tool [5], partial error masking [6], selective TMR [7], and Automatic Insertion of Partial TMR [8]

Triple modular redundancy (TMR) remains the standard fault-tolerance method for FPGAs [9]. It can be applied to circuits regardless of the function and of the logic cells used, and it often adds a minimal delay compared to other methods. TMR almost completely protects a circuit against a single fault, although voting logic may still be susceptible. However, it more than triples the circuit's size with a corresponding increase in power use. [10]. These advantages make TMR the most common basis for partial fault-tolerance.

Area-Constrained Partial Fault Tolerance is a technique that uses partial TMR to reduce the circuit area susceptible to faults even if the majority voters are not considered ideal, meaning that they can suffer faults also [3]. Ideal voters is an assumption often used for simplification in other methods because when a large subset of the circuit is being protected, the cross-sectional area of the majority voters is significantly smaller than the tripled area, perhaps by several orders of magnitude, and the rate of faults in the voters is considered to be negligible. This assumption is invalid in a fine-grained approach where the protected area and majority voters have comparable areas. ACPFT was originally designed to use several heuristics and metrics to determine a maximally fault-tolerant partition of a circuit's logic cells.

ACPFT maps well to genetic algorithms since it is similar to the familiar knapsack problem. In knapsack, there is a set of items, each with a weight and a value, and a knapsack that can hold a fixed weight. The optimization problem is to select a subset of items that can be carried in the knapsack with the maximum total value. The additional area of the FPGA relates to the knapsack, and the logic cells with their areas and sensitivities to faults relate to items with weight and value. However, the fault-tolerance problem is more complicated since the additional area required by each logic cell is not a constant value. It is a function of the other cells being protected. Previous research demonstrates that even simple genetic algorithms can create more fault-tolerant partitions than heuristic methods under common conditions, namely that the amount of additional resources available for fault tolerance is less than the size of the unprotected circuit [4].

2.2 Fundamentals of CUDA

CUDA is an extension to several common programming languages, prominently C and C++, which requires an NVIDIA-based graphics processor for execution. NVIDIA GPUs are widely deployed and thus represent a very common computing platform. For easy scalability, NVIDIA cards are designed around a generalized processing unit called a streaming multiprocessor (SM). This allows the performance of CUDA applications to scale based on the number and hardware implementation of the SMs contained on a given card. Details of NVIDIA GPUs can be found at [11].

Since the underlying hardware architecture of a GPU is drastically different than a CPU, algorithms must be crafted using several critical concepts in order to make efficient use of the GPUs computational power [12]. Since GPU-based cards have either a small cache system or none at all, they rely on massive thread parallelism to hide memory access latency by executing a different group of threads when one group block on a memory operation. Therefore, an algorithm must be able to extract sufficient parallelism from a problem to occupy the GPU's thread slots and mask this latency. Second, main memory accesses are efficient only when reading from contiguous memory locations. Data structures and memory accesses should be structured to use this coalesced pattern. Third, a GPU offers several different types of physical memory, such as the large global RAM, small local shared memory, constant memory, per-thread registers, and so on. Careful design should use the most appropriate memory type. Finally, an NVIDIA GPU executes threads in groups called warps. For each clock cycle, all of the threads in a warp or half-warp must either execute the same instruction or do nothing. When threads within a warp execute different code, called divergence, more and more threads will remain idle per clock cycle, and the processing power of the GPU is under-utilized.

With well-crafted algorithms designed with the above considerations, GPUs can potentially execute algorithms significantly faster than CPUs. GPUs can also solve problems using an estimated one tenth to one twentieth of the power required by traditional supercomputing systems [13], thereby reducing costs.

2.3 CUDA as a Platform for Evolutionary Algorithms

CUDA is already established as a popular platform for evolutionary computing. Examples can be found for simple genetic evolution [14] and differential evolution [15]. Ant colony optimization has also been explored [16-18] as well as particle swarm optimization [19-21].

In many evolutionary techniques, the execution time to evaluate the fitness function is a significant fraction of the algorithm's overall time, as is the case with ACPFT. Thus, even though there have been many implementations of evolutionary algorithms using CUDA, and the common operations are becoming well-understood, it is still crucial to explore efficient implementations of new, unique fitness functions and common evolutionary operators that support them.

3 Implementation

3.1 Genetic Algorithm Structure

The solution to the partial fault tolerance partitioning problem is coded as an ordered array of bytes in which each byte corresponds to a specific logic cell in the circuit. The byte's value is '0' if it is in the non-protected partition and '1' if it is in the protected partition. One byte is used per gene in the chromosome instead of one bit since other values are used temporarily during the constraint satisfaction check described later.

The algorithm randomly selects some chromosomes for mutation, choosing those with higher fitness functions proportionally more often. Each gene is examined for random mutation. Mutation results in the binary value being flipped. For crossover, chromosomes are selected in pairs. One gene is randomly selected as the location for single point crossover. Two new chromosomes are created from each pair selected. Mutation is not performed on chromosomes created by crossover.

The fitness function is simply the number of 1's in the chromosome, representing the number of logic cells that are in the protected subset. Previous research has shown that this correlates very highly with the actual amount of fault coverage provided. However, the complexity in ACPFT-GA is that the chromosome must represent a circuit that can fit within the available logic resources. Therefore, the chromosome is processed such that it represents a circuit with ACPFT correctly applied. In this format, a gene is '0' if it is unprotected, '1' if it is a protected and tripled cell, and '2' if the cell is tripled and connected to a majority voter. With this format, the amount of logic resources can be calculated and compared to those available. If the resulting circuit violates the constraints, the chromosome's fitness is set to a value lower than any possible valid chromosome. It is not culled since further evolution may result in a valid chromosome again.

3.2 Tool Chain

Each circuit is represented in a net list in the common EDIF format. ACPFT was written in Perl to accommodate reading and processing this input file and altering it for the partially protected output file. When using heuristics, the partitioning is processed with the Perl script. If the genetic evolution is selected, the ACPFT Perl script parses the EDIF net list outputs a condensed version of the net list in a text file. A C++ program then imports this formatted net list, performs the genetic evolution with or without using a GPU, and outputs the best chromosome. The ACPFT in turn uses this chromosome to partition the circuit and creates the proper modifications of the EDIF net list that can then be implemented on an FPGA. The script also generates a user constraint file to prevent the FPGA tools from removing the redundant logic cells. Currently, the ACPFT tools are run manually, but they are designed such that they could be easily inserted into the standard FPGA design flow and automated.

3.3 CUDA Implementation

The chromosomes are stored in a 2-dimenional array of characters in which each row corresponds to a complete solution. The array contains enough rows to hold all the members of the current generation and those created by mutation and crossover for the next. Once this entire array has been scanned to determine which chromosomes should be carried into the next generation in order of decreasing fitness function, the appropriate chromosomes are copied into a second identical array, and this second array is then the source of members for the next loop.

The GPU's constant memory was used to store many invariants of the net list, such as logic cell type, numbers of destination cells, and lists of destinations cells. It was also used to hold many kernel parameters that are fixed for a given evolution. None of these values are required to be in constant memory for the algorithm to function, so they could be moved to global memory if the circuit is too large for constant memory.

The pseudo code below shows the basic steps that are performed per generation by ACPFT-GA.

1. Generate array of pseudo-random values
2. Randomly select chromosomes for mutation and crossover
3. Create new chromosomes by mutation
4. Create new chromosomes by crossover
5. Calculate fitness function and validate constraints
6. Reorder the array of chromosomes.

CUDA has a random number library called cuRAND that has a CPU-only version. It is used in the CPU implementation so that both the CPU and GPU versions use the same pseudo-random sequences and generate identical output. Also, the cuRAND number generator is effective when generating large batches of values, so it is called once at the beginning of the loop to create all random values for that iteration.

In step 2, some random values are used to select chromosomes for steps 3 and 4. The fitness values of the chromosomes are summed, and each chromosome is assigned a range of values equal to its fitness. Each random number is scaled to the sum of fitness values, and then it is compared to the chromosomes' ranges to determine which is selected. In the GPU version, this operation is performed on the GPU. The kernel is written so that each block is 256 threads, each one converting one floating point random number to an integer corresponding to a chromosome. The number of blocks required is the ceiling of the number of chromosomes needed in the next two steps divided by 256. With the values tested, there were far too few blocks to fully occupy the card, but it is more efficient than transferring data to the CPU for computation.

In step 3, the algorithm selects a chromosome based on the indices from step 2, and it checks each gene for a mutation using random values still remaining from step 1. On the GPU implementation, one block is launched for each chromosome being mutated. The block size is set to 256, with each thread

checking every 256th gene for mutation. This allows the SM of a revision 2.0 GPU to hold 6 blocks, and the number of blocks/chromosomes needed to fully utilize the card is only a few dozen to a couple hundred.

The crossover step is similar to the mutation step, using indices from step 2 and the remaining random values from step 1. A pair of chromosomes is handled by one block, and each block uses 256 threads, again with each thread processing every 256th gene. Twice the number of chromosomes is needed to occupy the GPU than in step 3, but this number was easily reached.

Step 5 is by far the most complex. At this point, a chromosome contains only 1's and 0's representing the protected and non-protected subsets of logic cells. In ACPFT, each protected cell must be tripled, requiring three of that type of logic cell. For each tripled cell, all of the cells that use its output and are still classified as non-protected must be tripled and then combined with a majority voter. This step is performed first and is designed for as many coalesced accesses as possible, although some are unavoidable when examining a cell's destinations.

Once cells have been promoted to tripled and voted, each cell with a voter is examined to see if all destination are tripled or voted. If so, the logic cell can be converted to a tripled cell, increasing the size of the protected subset and actually freeing some resources used for voters. This step also has coalesced and non-coalesced accesses.

After this step, the fitness function is calculated with a reduction from the CUDA thrust library. The sum consists of the sensitivities for all cells that are in the protected subset, ignoring cells that are single or tripled. Next, the constraint condition is checked. Each logic cell adds a count to the logic cells used based on its state. Single cells add one to the like type, tripled cells add three to the like type, and voted cells add three to the like type and one to the type used for voters. These counts are contained in shared memory and require atomic addition instructions to avoid races.

Once all of the logic cells are accounted for, one thread compares the needed resources to the available resources. If the constraints aren't met, the fitness value is adjusted to 1.0, so that the invalid chromosome still has a small chance of being selected in the next generation. Further mutation and crossover may again result in a valid cell.

The CUDA thrust library is used in step 6. The fitness values are sorted and the new chromosome order is determined using these optimized sorting functions. Another thrust function calculates the prefix sum used for the chromosomes ranges. Finally, a kernel uses the new order to copy the best chromosomes from the current chromosome array into the second array, and the pointers to these arrays are swapped in preparation for the next generation.

4 Testing and Results

ACPFT-GA was tested using the *alu4*, *apex2*, and *pdc* circuits from the ACM/SIGDA "Big 20" benchmarks. These circuits were chosen since they show a range of circuit sizes with 597, 1056, and 1328 logic cells respectively. They have also been used in previous research, and data exists for comparisons.

As with previous ACPFT experiments, the performance given different amounts of available resources is accomplished by creating an array of theoretical FPGAs of varying sizes. The number of logic cells in each circuit is used as a "perfect-fit" FPGA. Larger devices are emulated by increasing the number of each resource by a constant multiplier and rounding down. This method created up to 23 theoretical FPGAs, from 10% to 230% in increments of 10%. Between 210% and 230% depending on the test circuit, there were sufficient resource for full TMR, and partial fault-tolerance would no longer be necessary.

The genetic algorithm parameters were selected to match those used in [4]. The test used a set of 4096 chromosomes. For each circuit, the mutation factor was the reciprocal of the number of logic cells. In each generation, the top 256 chromosomes were carried over into the next generation. 1920 were selected using elitism for mutation, and each gene was checked for a mutation. The remaining 1920 chromosomes were generated by crossover. Mutations were not applied to chromosomes created through crossover. Like the previous work, each initial chromosome was initialized to a string of "0"s, representing a fully unprotected circuit. Three more experiments were performed using the output of a heuristic method from previous research to initialize the chromosomes. The fanout method was chosen since it yielded very good results and had a low execution time. For the second experimental setup, all additional resources were supplied to the fanout method, and then the result was refined with the genetic algorithm. In the third and fourth setups, the fanout method was supplied 10% fewer and 20% fewer resources than available respectively. In these cases, there were still some unallocated logic cells when the genetic algorithm was applied.

The test computer used a Core i7 processor at 2.8 GHz, 6 GB of RAM, a GTX 480 graphics card, and CUDA SDK 4.0. ACPFT-GA was run ten times on each simulated FPGA using just a CPU and then using the GPU. Each test was allowed to run for 1000 generations. The times required for mutation, crossing, and calculating the fitness functions and constraint conditions were logged for all runs.

The mutation time, crossover time, fitness function and constraint checking time, and total execution time are shown in the following tables. Times are shown for the CPU-only version, the GPU accelerated version, and the speedup of the GPU version relative to the CPU version. The data for *alu4*, *apex2*, and *pdc* are shown in Table 1, Table 2, and Table 3 respectively for the tests that begin with no initialization, i.e. all available resources are unused and selected only by the genetic algorithm. The data for tests in which 20% of the resources are left unused, and the chromosomes are initialized with the output of ACPFT using fanout are shown in Table 4, Table 5, and Table 6 for *alu4*, *apex2*, and *pdc* respectively. The data for the other two tests are not shown due to space limitations, but the results are very comparable to the 20%

used tests. These tables show results from 10% additional resources to 230% additional resources.

The data shows several patterns. First, the amount of time required for mutation and crossover remains fairly fixed for each circuit over the range of additional resources. This is expected, since the work performed for the mutation and crossover steps depends on the number of genes and the number of chromosomes. From circuit to circuit, the differences in mutation times and crossover times for both versions were smaller than the difference in circuit sizes. This indicates that these steps are communication bound, as is expected.

The fitness and constraints checking execution time shows much more variation. The pattern of this variation is shown in Figure 1 for the *pdc* circuit with no initialization accelerated with the GPU, and it is representative of graphs of other tests.
This graph demonstrates that the amount of time required for the fitness function is very dependent on the amount of additional resources made available. This pattern is logical. To evaluate the fitness function, each cell that is within the protected subset must be examined to triple and vote its output cells, followed by examing voters to see if they can be removed. As the amount of available resources increases, the number of cells within the protected partition increases, and thus the execution time required also increases.
The data shows that the mutation and crossover speedups are comparable between circuits. The tables also demonstrate that the crossover time consumes a few percent of the total time, the mutation time is usually within 10% to 20% of the execution time (with lower percentages as more resources are made available), and the fitness function consumes the majority of the processing time. Therefore, the total speedup depends largely on the speedup of the fitness function. The speedups between circuit is also very similar. All three circuits were implemented with the same time of logical device and had about the same average fanout. Therefore, the amount of work per logic cell in the protected subset was roughly the same for all three circuits.

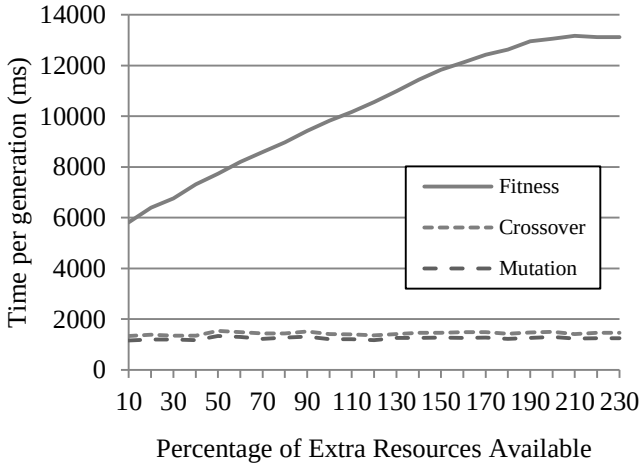


Figure 1 Per Generation Execution Time for the *pdc* Circuit using a GTX 480 and 20% Unused Additional Resources

Table 1 Performance of *alu4* with no initialization

	Mutation	Crossover	Fitness and Constraints	Total
Min. time w/CPU	15,874.2 ms	5,874.8 ms	55,103.7 ms	77,068.7 ms
Max. time w/CPU	17,267.5 ms	7,044.1 ms	89,340.1 ms	113,767.3 ms
Ave. time w/CPU	16,641.1 ms	6,478.9 ms	76,733.0 ms	99,968.7 ms
Min. time w/GPU	856.4 ms	123.7 ms	3,011.3 ms	5,081.7 ms
Max. time w/GPU	1,134.8 ms	185.1 ms	7,093.1 ms	19,412.1 ms
Ave. time w/GPU	1,068.7 ms	153.9 ms	5,504.3 ms	12,742.3 ms
Min. Speedup	14.0	32.1	12.4	13.4
Max. Speedup	20.0	54.1	17.3	17.5
Ave. Speedup	15.6	42.5	14.3	15.1

Table 2 Performance of *apex2*with no initialization

	Mutation	Crossover	Fitness and Constraints	Total
Min. time w/CPU	25,186.7 ms	7,774.0 ms	108,212.6 ms	144,558.5 ms
Max. time w/CPU	26,760.9 ms	8,917.8 ms	158,688.5 ms	194,473.5 ms
Ave. time w/CPU	27,102.6 ms	8,420.2 ms	142,759.2 ms	179,140.1 ms
Min. time w/GPU	1,158.2 ms	154.6 ms	4,481.6 ms	8,743.7 ms
Max. time w/GPU	1,340.5 ms	232.3 ms	11,750.8 ms	47,051.5 ms
Ave. time w/GPU	1,243.5 ms	194.2 ms	8,850.8 ms	29,703.4 ms
Min. Speedup	19.5	36.8	12.9	14.0
Max. Speedup	22.4	56.4	23.3	24.2
Ave. Speedup	21.8	43.8	17.1	18.2

Table 3 Performance of *pdc* with no initialization

	Mutation	Crossover	Fitness and Constraints	Total
Min. time w/CPU	31,728.3 ms	8,887.0 ms	131,546.4 ms	172,293.0 ms
Max. time w/CPU	32,522.6 ms	9,804.8 ms	194,660.3 ms	236,821.8 ms
Ave. time w/CPU	32,185.7 ms	9,396.8 ms	174,245.2 ms	215,949.1 ms
Min. time w/GPU	1,243.5 ms	177.4 ms	4,865.0 ms	10,802.9 ms
Max. time w/GPU	1,398.7 ms	245.9 ms	16,053.1 ms	70,709.3 ms
Ave. time w/GPU	1,340.3 ms	206.6 ms	12,231.8 ms	42,948.1 ms
Min. Speedup	23.0	38.5	12.0	13.3
Max. Speedup	25.7	53.9	27.0	27.1
Ave. Speedup	24.0	45.9	15.1	16.4

Table 4 Performance of *alu4* initialized with fanout heuristic and 20% area free

	Mutation	Crossover	Fitness and Constraints	Total
Min. time w/CPU	16,219.7 ms	6,019.2 ms	60,460.7 ms	82,970.3 ms
Max. time w/CPU	20,661.1 ms	9,091.9 ms	93,772.2 ms	123,792.7 ms
Ave. time w/CPU	19,772.6 ms	8,386.4 ms	84,708.5 ms	113,263.4 ms
Min. time w/GPU	1,039.0 ms	132.1 ms	3,554.2 ms	6,502.0 ms
Max. time w/GPU	1,052.8 ms	154.2 ms	6,799.4 ms	11,178.7 ms
Ave. time w/GPU	1,044.1 ms	139.3 ms	5,438.7 ms	8,500.5 ms
Min. Speedup	15.5	43.4	13.6	15.2
Max. Speedup	19.7	66.0	19.2	20.3
Ave. Speedup	18.9	60.2	16.0	17.3

Table 5 Performance of *apex2* initialized with fanout heuristic and 20% area free

	Mutation	Crossover	Fitness and Constraints	Total
Min. time w/CPU	25,970.9 ms	8,020.1 ms	120,902.6 ms	156,958.0 ms
Max. time w/CPU	28,001.2 ms	9,104.0 ms	146,805.2 ms	182,581.4 ms
Ave. time w/CPU	26,439.2 ms	8,570.1 ms	137,966.6 ms	173,902.9 ms
Min. time w/GPU	1,109.1 ms	111.0 ms	5,639.5 ms	13,036.7 ms
Max. time w/GPU	1,298.5 ms	158.0 ms	11,483.9 ms	22,720.0 ms
Ave. time w/GPU	1,164.5 ms	135.2 ms	8,901.9 ms	15,913.5 ms
Min. Speedup	20.5	55.9	12.5	14.0
Max. Speedup	23.9	78.6	21.6	22.9
Ave. Speedup	22.7	63.9	16.1	17.6

Table 6 Performance of *pdc* initialized with fanout heuristic and 20% area free

	Mutation	Crossover	Fitness and Constraints	Total
Min. time w/CPU	31,476.9 ms	8,633.9 ms	145,464.6 ms	187,116.4 ms
Max. time w/CPU	32,764.1 ms	9,961.1 ms	186,007.6 ms	228,721.8 ms
Ave. time w/CPU	32,211.5 ms	9,339.5 ms	172,327.6 ms	214,013.6 ms
Min. time w/GPU	1,277.9 ms	145.2 ms	8,104.6 ms	14,250.6 ms
Max. time w/GPU	1,325.3 ms	197.7 ms	15,348.0 ms	29,715.3 ms
Ave. time w/GPU	1,296.7 ms	173.9 ms	12,094.0 ms	22,916.1 ms
Min. Speedup	23.9	43.7	12.1	13.6
Max. Speedup	25.5	65.8	17.9	19.6
Ave. Speedup	24.8	54.2	14.7	16.1

5 Conclusions

This research presents a significant acceleration of the partial fault tolerance method area-constrained partial fault tolerance using genetic evolution by employing NVIDIA CUDA to execute the algorithm on massively parallel graphical processing units. This speed up allows this method to be much more efficiently applied to larger circuits, and it will benefit from additional acceleration as the processing power of graphics processors tracks that of reconfigurable logic devices.

6 References

[1] J. Lach, *et al.*, "Efficiently Supporting Fault Tolerance in FPGAs," presented at the ACM International Symposium on FPGAs, 1998.

[2] F. L. Kastensmidt, *et al.*, "On the optimal design of triple modular redundancy logic for SRAM-based FPGAs," in *Design, Automation and Test in Europe*, 2005, pp. 1290-1295.

[3] D. L. Foster and D. M. Hanna, "Maximizing Area-Constrained Partial Fault Tolerance in Reconfigurable Logic," presented at the Proceedings of the 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, California, USA, 2010.

[4] D. L. Foster, "Using Genetic Algorithms for Subset Selection for Partial Fault Tolerance in Reconfigurable Logic," in *The 2011 International Conference on Genetic and Evolutionary Methods*, Las Vegas, NV, 2011.

[5] B. Pratt, *et al.*, "Improving FPGA Design Robustness with Partial TMR," presented at the 12th NASA Symposium on VLSI Design, Coeur d'Alene, Idaho, 2005.

[6] K. Mohanram and N. A. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits," in *Test Conference, 2003. Proceedings. ITC 2003. International*, 2003, pp. 893-901.

[7] P. K. Samudrala, *et al.*, "Selective triple Modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs," *Nuclear Science, IEEE Transactions on*, vol. 51, pp. 2957-2969, 2004.

[8] O. Ruano, *et al.*, "A Methodology for Automatic Insertion of Selective TMR in Digital Circuits Affected by SEUs," *IEEE Transactions on Nuclear Science*, vol. 56, pp. 2091-2102, August 2009 2009.

[9] H. Quinn, *et al.*, "Domain Crossing Errors: Limitations on Single Device Triple-Modular Redundancy Circuits in Xilinx FPGAs," *IEEE Transactions on Nuclear Science*, vol. 54, pp. 2037-2043, 2007.

[10] F. Lima, *et al.*, "Designing fault tolerant systems into SRAM-based FPGAs," presented at the Design Automation Conference, 2003.

[11] NVIDIA Corp. (2011, Mar. 8). *NVIDIA CUDA C Programming Guide Version 4.1*. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[12] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: a Hands-On Approach*: Morgan Kaufmann Publishers, 2010.

[13] NVIDIA. (2010, Nov. 23). *Tesla C2050/C2070 GPU Computing Processor Overview*.

[14] A. Munawar, *et al.*, "Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework," *Genetic Programming and Evolvable Machines*, vol. 10, pp. 391-415, 2009.

[15] P. Krömer, *et al.*, "Many-threaded implementation of differential evolution for the CUDA platform," presented at the Proceedings of the 13th annual conference on Genetic and evolutionary computation, Dublin, Ireland, 2011.

[16] J. M. Cecilia, *et al.*, "Parallelization Strategies for Ant Colony Optimisation on GPUs," in *14th International Workshop on Nature Inspired Distributed Computing*, Anchorage, AK, USA, 2011.

[17] S. Tsutsui and N. Fujimoto, "ACO with tabu search on a GPU for solving QAPs using move-cost adjusted thread assignment," presented at the Proceedings of the 13th annual conference on Genetic and evolutionary computation, Dublin, Ireland, 2011.

[18] H. Bai, *et al.*, "MAX-MIN Ant System on GPU with CUDA," presented at the Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control, 2009.

[19] L. Mussi, *et al.*, "GPU-based asynchronous particle swarm optimization," presented at the Proceedings of the 13th annual conference on Genetic and evolutionary computation, Dublin, Ireland, 2011.

[20] H. Zhu, *et al.*, "Paralleling Euclidean Particle Swarm Optimization in CUDA," presented at the Proceedings of the 2011 4th International Conference on Intelligent Networks and Intelligent Systems, 2011.

[21] B. Rymut and B. Kwolek, "GPU-supported object tracking using adaptive appearance models and particle swarm optimization," presented at the Proceedings of the 2010 international conference on Computer vision and graphics: Part II, Warsaw, Poland, 2010.

RADIC: A FaultTolerant Middleware with Automatic Management of Spare Nodes*

Hugo Meyer[1], Dolores Rexachs[2], Emilio Luque[2]
Computer Architecture and Operating Systems Department, University Autonoma of Barcelona, Bellaterra, Barcelona, Spain
[1] hugo.meyer@caos.uab.es, [2]{dolores.rexachs, emilio.luque}@uab.es

Abstract— *As the mean time between failures has decreased, applications should be able to handle failures avoiding performance degradation if possible. This work is focused on a decentralized, scalable, transparent and flexible fault tolerant architecture named RADIC an acronym for Redundant Array of Distributed and Independent Controllers. As MPI is a de facto standard used for communication in parallel computers, RADIC have been included into it. RADIC's default behavior is to restart failed processes on already used nodes, overloading them. Moreover it also could be configured to keep the initial process per node ratio, maintaining the original performance in case of failures. In this paper we propose a transparent and automatic management of spare nodes in order to avoid performance degradation and to minimize the mean time to recovery – MTTR when using them. Our work provides transparent fault tolerance for applications that are written using the MPI standard. Initial evaluations show how the application performance is restored as it used to be before a failure, minimizing the MTTR by managing faults automatically.*

Keywords – RADIC, MPI, Fault Tolerance, Decentralized, Spare Nodes, Uncoordinated Checkpoint.

1 Introduction

Considering the many long-running parallel applications that are executed on High Performance Computer – HPC-clusters and the increase in the failure rate [1] on these computers, it becomes imperative to make these applications resilient to faults.

Hardware failures may cause unscheduled stops to applications. If there are not any fault tolerant mechanisms to prevent it, these applications will have to be re-executed from the beginning. If a fault tolerant mechanism is used, failures could be treated. In such environment an automatic and application transparent fault tolerance mechanism is desirable. It could also reduce the complexity of applications development. Failure treatment and management are crucial to maintain the performance of HPC applications that are executed over several days.

One of the most commonly used approaches to deal with failures in HPC parallel applications is the rollback-recovery approach based on checkpoint and restart protocols. Rollback-recovery protocols periodically save processes states in order to rollback in case of faults.

Checkpoints could be performed using a coordinated or uncoordinated checkpointing protocol. Coordinated checkpointing protocols create a consistent set of checkpoints by stopping all the processes in the parallel application in a consistent state and then taking a snapshot of the entire application. This approach minimizes the overhead of fault free execution, but in case of faults, all processes (even those that have not failed) must rollback to the previous consistent saved state. All the computation time used to progress the parallel application execution before the fault and after the last snapshot is loosed.

In uncoordinated checkpointing protocols, each process is checkpointed individually, and it could happen in different moments of the execution. Thus, there is not a global consistent state. The advantage of this method is that in case of faults only the affected processes must rollback. In order to avoid the domino effect [2], this approach should be combined with an event logging protocol.

When a parallel application is executed, we usually seek for executions with an optimal amount of resources to maximize the speedup or efficiency. When a failure occurs and the application loses some resources all the initial tuning effort is loosed.

In this paper we present new RADIC [3] enhancements to avoid performance degradation when failures occur. The objective is achieved using automatic spare nodes management to maintain the initial amount of resources when node failures occur. We also try to minimize the MTTR after a failure is detected by managing faults without human intervention. For that reason, every fault tolerant tasks and decisions are made automatically. The RADIC architecture has been integrated into the Open MPI library to allow execution of real scientific parallel applications and to be application-transparent.

Our approach considers the consequences that node failures bring to parallel applications. A physical failure affects computing components. If these components are not replaced properly there is a loss in computational capacity.

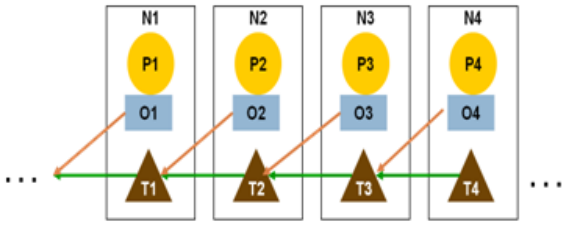


Figure 1. RADIC components

* This research has been supported by the MICINN Spain under contract TIN2007-64974, the MINECO (MICINN) Spain under contract TIN2011-24384, the European ITEA2 project H4H, No 09011 and the Avanza Competitividad I+D+I program under contract TSI-020400-2010-120. This paper is addressed to the **PDPTA'12** Conference

Running a parallel application with fewer resources than the optimal, causes this degradation.

This work is presented and divided as follow: section 2 describes the RADIC architecture, its components and how it operates to protect an application against failures. In section 3 we introduce the related work on fault tolerant systems. The section 4 presents the integration of RADIC into the Open MPI library to provide user-transparent fault tolerance. Next, section 5 illustrates the initial results obtained with the described implementation. Finally, section 6 presents the conclusions and future lines.

2 RADIC Architecture

RADIC [3] is a fault tolerant architecture for message passing systems based on *rollback-recovery* techniques. These techniques rely on uncoordinated checkpoint protocol combined with a receiver based pessimistic event log [4]. The approach that was chosen does not need any coordinated or centralized action or element to carry out their fault tolerance tasks and mechanisms, so application scalability depends on the application itself.

The RADIC architecture acts as a fault tolerant layer between the MPI standard and the parallel machine (fault probable). This fault tolerant layer provides a fault-resilient environment for parallel application even when the application runs over a fault-probable parallel machine.

Our work is focused on providing an application-transparent fault tolerant middleware within a message passing library, specifically, Open MPI [5].

Critical data such as checkpoints and event logs are stored in a different node than the one in which the process is running. Processes that were residing in a failed node will be restarted in another node from their latest checkpoint, and will consume the event log in order to reach the before fault state. RADIC policies provide a transparent, decentralized, scalable and flexible fault tolerance solution.

2.1 RADIC components

RADIC provides fault tolerance based on two main components: *protectors* and *observers*. In the Figure 1 we illustrated computing nodes (Ny), application process (Px), the protectors (Ty), and the observers (Ox) where the sub-index *x* represents the process number and *y* represents the node number. Protectors and observers work together with the aim of building a distributed fault tolerant controller. Both components are described below:

- *Observers*: are responsible of process monitoring and fault masking. Each application process has one observer attached to it. The observers performs event logging of received messages in a pessimistic manner, they also take periodic checkpoints of the process to which it is attached. Checkpoints and logging data are sent and stored in their protectors located in another node (Figure 1). During recovery, the observers are in charge of processing with the event log, replaying them in order to reach the same state before fault.
- *Protectors*: on each node there is a protector running, their main function is to detect node failures via a heartbeat/watchdog protocol. Protectors also store checkpoints and event logs sent by observers. When a failure occurs, the protector has to restart the failed processes that it protects; they also have to reestablish the heartbeat/watchdog protocol since it gets broken due to node failures.

2.2 RADIC Operation

Fast failure detection is one of RADIC priorities, since it is one of the variables that affect the MTTR. RADIC first detection mechanism is a heartbeat/watchdog protocol that allows protectors to learn about neighbor's protectors faults.

As every communication goes through the observers, they have absolutely control of messages exchange between peers. Observers can also detect and mask faults. Each protector maintain a data structure called *radictable*, where each entry (an entry per process exists in the application) of the structure is composed of a process id, the URI of the process, URI of its protector, and a unique clock of received and sent messages. When a process fails and get restarted, the observers consult the *radictable* in order to find about the node where the process has been recovered by asking the process's protector. The protectors updated the *radictable* on demand when they identify any processes failures.

In the Figure 2a it is possible to see a fault free execution using RADIC without spare nodes support.

When a failure occurs (Figure 2b), the parallel application execution will continue with one less node. The node failure is detected by the heartbeat/watchdog mechanism. After the failure, the heartbeat/watchdog mechanism is reconstructed, and T4 indicates T2 as the new protector of P4 (Figure 2c). O4 needs to take a checkpoint of P4, because its latest checkpoint gets lost when T3 fails. T2 restarts and re-executes P3(Figure 2d), and also will indicate that the new protector of P3 is T1. Then O3 will take a checkpoint of P3 and send the data to T1. Finally, O3 we erase old message logs.

The protectors have two operating modes: active or passive. Active is when they form part of the detection scheme and there are some application processes running on its node (all nodes of Figure 2). Protectors may be in a passive state when they are running in a spare node, this is a low consumption state (to avoid node and network overload).

2.3 Spare Nodes in RADIC

When a failure occurs and the failed process is restarted in the same node its protector is running, if this node already has application processes running on it, the node becomes overloaded. This could slow down the execution of both

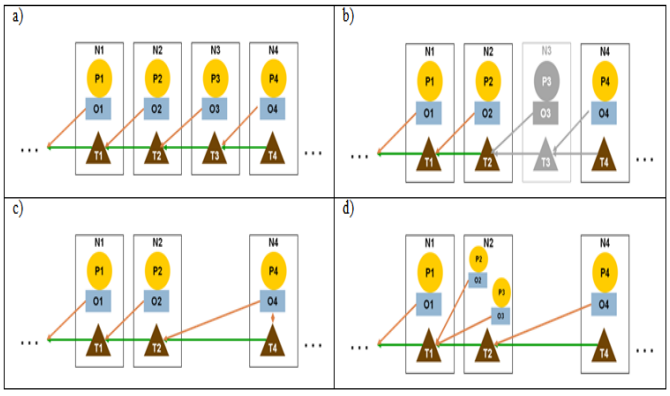


Figure 2. a) Fault free execution. b) Failure in Node 3.
c) Heartbeat/watchdog restoration and assignation of a new protector to P4.
d) Restart of the process P3 in node N2.

processes. As a consequence of this, the performance of the entire application could be affected, increasing its execution time.

One method to maintain the initial performance in such a scenario is to use spare nodes to restart failed processes [6] instead of overloading the non-failed nodes. Spare nodes are that initially are not used by the parallel application.

In the Figure 3a, we can observe the execution of a parallel application using 4 nodes and having 1 spare node (NS). When a failure occurs in the node N3 (Figure 3b) the protector T2 will detect the failure of protector T3 and it consult a table to find the information about the spare nodes location and state (*sparetable*).

The *sparetable* (Table 1) is replicated among all protectors. Spares are assigned as failures occur, and the replicated information is updated on demand, so all the operations are made in a decentralized and transparent manner. Eventually, these tables could be outdated, however it does not affects the RADIC operation, since this information will be checked before using any spare.

After consulting its *sparetable* the protector T2 confirms the availability of the spare NS (Figure 3c) and if it is available T2 transfer the latest checkpoint and event logging data of process P3 to NS (Figure 3d). Finally, the protector TS restart P3 and become an active protector by joining the heartbeat/watchdog protection scheme (Figure 3e).

3 Related Work

Many proposals have been made to provide fault tolerance for message passing applications. Most strategies are based on a coordinated checkpointing approach or an uncoordinated checkpointing strategy combined with a logging mechanism.

Currently, there are several checkpoint-restart tools available. We can highlight BLCR (*Berkeley Lab's Checkpoint/Restart*) [7] and DMTCP (*Distributed MultiThreaded Checkpoint*) [8]. DMTCP works at user space and BLCR works at kernel level. BLCR is one of the most used libraries to provide fault tolerance in parallel systems. To use BLCR in parallel applications, MPI libraries should at least reopen communication channels after restart [9].

Table 2 highlights the features of three of the most popular fault tolerant frameworks integrated into MPI libraries and our approach. Most solutions use a centralized storage. However, due to scalability reasons, it is desirable to avoid any centralized element.

Our approach differs from MPICH-V2 [10] because we do not use any centralized storage because with RADIC, every computing node could stores critical data from process residing in another node. Also we use a pessimistic receiver based logging protocol. MPICH-V2 is now a deprecated implementation.

MPICH-VCL is designed to reduce overhead during fault free execution by avoiding message logs. It is based on Chandy-Lamport algorithm [11]. MPICH2-PCL [12] uses a blocking coordinated checkpointing protocol.

LAM-MPI [13] is previous to Open MPI. It modularizes a *checkpoint/restart* approach to allow the usage of multiple *checkpoint/restart* techniques. The implementation supports communications over TCP and Myrinet in combination with *BLCR* and *SELF* checkpointing operations. *LAM-MPI* uses a coordinated checkpoint approach and needs a communication thread between the checkpoint/restart system and the process *mpirun* to schedule checkpoints.

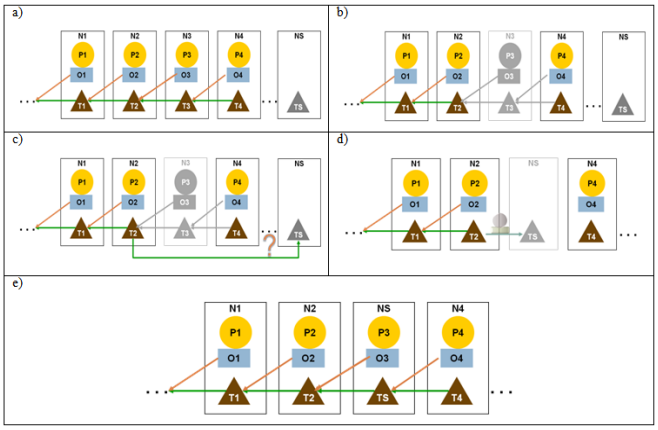


Figure 3. RADIC with spare nodes. a) Execution before failure with one spare node. b) Failure in node 3. c) The protector T2 check availability of spare node NS. d) Protector T2 transfer the checkpoint of process P3 to spare node NS. e) Protector TS restart process P3 and also the communications.

The current *checkpoint/restart* implementation of the Open MPI library [9] aims to combine the best features from these methods described above. The implementation uses a distributed checkpoint/restart mechanism where each checkpoint is taken independently, but coordination is needed to make a consistent global state, which requires the interruption of all processes at the same time.

Another work that has become important is the Coordinated Infrastructure for Fault Tolerant Systems –CIFTS- [14]. It is a framework that enables system software components to share fault information with other components to take some action in order to get adapted to faults. The main difference with our proposal is that we deal with faults automatically and transparently to applications. This allows us to reduce the MTTR.

4 RADIC in MPI

The first prototype of RADIC was called RADICMPI [15] and it has been developed as a small subset of the MPI standard. As a message passing library is very limited. As this implementation does not have all the MPI primitives, it cannot execute many of the scientific applications available

Table 1. Sparetable

Spare Id	Address	Observers
0	Node5	1
1	Node6	0
…	…	…

RADICMPI does not consider collective operations and other complex functions that many applications use. For that reason, instead of extending the prototype to comply the MPI standard, we decided to integrate the RADIC architecture into a well-established MPI implementation. It allows the correct execution of any MPI application using the fault tolerance policies and mechanisms of RADIC (Section II).

In the next paragraphs we will explain some important features of the integration of RADIC into Open MPI.

4.1 Open MPI Architecture

A depth research about the inclusion of RADIC in Open MPI has been made in [16]. The implementation is named

RADIC-OMPI and integrates the basic protection level of RADIC. It does not include spare nodes management.

Open MPI architecture has been already described in [5]. For that reason, in this paper we will focus only on the components relevant to the RADIC integration.

The Open MPI frameworks are divided in three groups that are: *Open MPI (OMPI)* which provides the API to write parallel applications; *Open Run-Time Environment (ORTE)* which provides the execution environment for parallel applications; and *Open Portable Layer (OPAL)* which provides an abstraction to some operating system functions.

To launch a given parallel application, an ORTE daemon is launched in every node that takes part in the parallel application. These daemons communicate between them to create the parallel runtime environment. Once this environment is created the application processes are launched by these daemons. Every process exchange information about communication channels during the Module Exchange (MODEX) operation which is an all-to-all communication. The *protector* functionalities have been integrated into the ORTE daemon because in Open MPI there is always one daemon running in each node, wich fits the protector requirements.

Table 2. Fault tolerant MPI libraries.

Name	FT Strategy	Detection and Recovery
MPICH-V2	- Uncoordinated Ckpt. - Sender based pessimistic log. - Centralized storage.	- Automatic.
MPICH-VCL	- Coordinated Ckpt. - Chandy-Lamport Algorithm. - Centralized storage.	- Automatic.
Open MPI	- Coordinated Ckpt. - Centralized storage.	- Fault Detection and safe stop. - Manual recovery.
RADIC	- Uncoordinated Ckpt. - Pessimistic Receiver based Log. - Distributed storage.	- Automatic and application transparent.

OMPI provides a three-layer framework stack for MPI communication:

- *Point-to-point Management Layer (PML)* which allows wrapper stacking. The **observer**, because of its behavior, has been implemented as a PML component; this ensures the existence of one observer per application process.
- *Byte Transfer Layer (BTL)* that implements all the communication drivers.
- *BTL Management Layer (BML)* that acts as a container to the drivers implemented by the BTL framework.

The Open MPI implementation provides a framework to schedule checkpoint/restart requests. This framework is called *Snapshot Coordinator (SnapC)*. The generated checkpoints are transferred through the *File Manager (FileM)* framework. All these communications to schedule and manage the transferring of the checkpoint files are made using the *Out of Band (OOB)* framework.

4.2 RADIC Implementation

To define the initial heartbeat/watchdog fault detection protection scheme and protection mapping a simple algorithm is used: each observer sets his protector as the next logical node, and the last node sets the first one as its protector.

All protectors should fill the *radictable* before launching the parallel application and update it with new information when failures occur. The update of the *radictable* does not require any collective operation. Thus many protectors could have an outdated version of the *radictable*. However, the *radictable* will be updated further on demand, when observers try to contact restarted processes.

Regarding to the fault tolerances mechanism and their integration into Open MPI, the following observations can be made:

- *Uncoordinated checkpoints:* each process performs its checkpoints through a checkpoint thread. Checkpoints are triggered by a timer (*checkpoint interval*) or by other events. Before a checkpoint is made, to ensure that there is no in transit messages all communication channels are flushed and remain unused until the checkpointing operation finishes. After a checkpoint is made, each process transfers their checkpoint files using the *FileM* framework and then the communication within processes are allowed again.
- *Message Log:* since the observer is located in the PML framework, it ensures that all communications through it are logged and then transferred to the correspondent protector. The protector only confirms a message reception after the message has been saved. Messages are marked as received by the remote process after the receiver and its protector confirm the message reception (pessimistic receiver based log).
- *Failure detection mechanism:* failures are detected when communications fails; this mechanism requires the modification of lower layers to raise errors to the PML framework where the faults are managed. It avoids application stops. A heartbeat/watchdog mechanism is also used. The protectors send heartbeats to the next logical node and the receiver protector resets the *watchdog* timer after reception.
- *Failure management:* the default behavior of the library is to finalize when a failure occurs (*fail-stop*) Hence RADIC needs to mask failures to continue execution and avoid fault propagation to the application level. When a protector finds out about a failure, the restarting operation is initiated.
- *Recovery:* the recovery is composed of three phases. In the first one, a protector restores the failed process from its checkpoint with its attached observer. Then the restored observer sets its new protector, re-executes the process while consuming the event logging data and then takes a checkpoint. Finally, the process execution is resumed after its checkpoint is sent to its new protector, just to ensure its protection. Protectors involved in the fault also reestablish the protection mechanism. We consider the recovery as an atomic procedure.
- *Reconfiguration:* when the recovery ends, the communications have to be restored. To achieve this

goal the lower layers of Open MPI must be modified to redirect all the communications to the new address of the process. To avoid collective operation this information is updated on demand or by a token mechanism.

4.3 Proposal: Spare Nodes Management in Open MPI

An important aspect that has to be considered when running parallel applications is the performance. The previous implementation of the RADIC architecture [16] allows the successful completion of parallel applications even in presence of failures. However, it does not consider the management of extra resources to replace failed nodes. Including the spare nodes management into RADIC, the applications will not only end correctly but also will avoid performance degradation due to loss of computational resources.

Our proposal is not restricted on avoiding performance lost, we also propose a mechanism for automatically select spare nodes and include them on the parallel environment domain without user intervention. By doing the spare nodes management transparently and automatically, we minimize the MTTR.

When including spare nodes into the RADIC architecture, the restarting and reconfiguration are the most affected mechanisms. To reconfigure the system, a deterministic algorithm to find restarted processes is needed.

When using RADIC without spare nodes (Figure 2), failed processes are restarted in their protectors. If an observer tries to reach a relocated failed process, it will take a look at its *radictable* to find the old protector of the failed process (this information may be outdated). Then, the observer will ask about that process. The old protector will say that it is no longer protecting such a process, and will point who is the new protector (Figure 2).

If a failure occurs and there are spare nodes available, the spare will be included into the parallel environment domain and failed processes should be restarted in it. The heartbeat/watchdog mechanism will be reestablished and the involved protectors will update their *radictable* and *sparetable* (Table 1).

Considering Figure 3e, if process P1 wants to reach P3, O1 will ask T2 about P3. T2 will point that P3 is residing in the spare NS. Then O1 will tell T1 to update its radictable and its sparetable and P1 will finally try to contact P3. The process described above is distributed and decentralized, and each process will do it only when it is strictly necessary, avoiding the costly *Module Exchange* (MODEX) collective of Open MPI.

The main problem when restarting a process in another node is that we need an ORTE daemon running in that node to adopt the new process as a child. Moreover, all future communication with the restarted process needs to be redirected to its new location. For that reason, ORTE daemons are launched even in spare nodes, but no application process is launched on it until it be required as a spare node.

An additional problem that must be addressed is that a sender observer must not consider as a failure the lack of communication with other processes when the receiver process is doing a checkpoint or is restarting. The sender observer will fail to communicate, and will consult the receiver's protector to find about the state of the receiver. The protector will indicate that the process is checkpointing or restarting, and the communication will be retried later.

The *radictable* and *sparetable* were included inside the job information structure (*orte_jmap_t*). When the parallel application starts, each protector (ORTE daemon) populates its *radictable* and its *sparetable*. The *radictable* and *sparetable* are updated (on demand) when a protector notices that a process has restarted in another place.

If the application runs out of spares, the default mechanism of RADIC is used (Figure 2).

5 Experimental Results

A fault tolerant architecture, generally, introduces some kind of overhead in the system it is protecting. These overheads are generally caused by replication in some of its forms. The overheads introduced by RADIC are mostly caused by the uncoordinated checkpoints and the pessimistic log mechanism as it has been showed in [16].

Failures may cause degradation because of the loss of the computational capacity if there are no spare nodes available. The experimental evaluation that has been done tries to shows how fast is the failure detection and recovery mechanisms of our proposal, and how fast it can include automatically spare nodes into the parallel environment in order to avoid the impact on the performance of applications when resources are loosed.

We present experimental results using three different benchmarks: a static matrix multiplication benchmark, the LU benchmark that is part of the *NAS Parallel Benchmarks* (*NPB*) [17] and the SMG2000 application [18].

The matrix multiplication application is modeled as a *master/worker* application, the master sends the data to the workers only at the start, and collects the results when the application finalizes. Each application process is assigned to one core during normal execution. The matrix multiplication implemented has few communications (only at the beginning and at the end).

Experiments have been made using a *Dell PowerEdge M600* with 8 nodes, each node with 2 quad-core Intel® Xeon® E5430 running at 2.66 GHz. Each node has 16 GBytes of main memory and and a dual embedded Broadcom® NetXtreme IITM 5708 Gigabit Ethernet. RADIC have been integrated into version 1.7 of Open MPI.

Our main objective is to depict the application

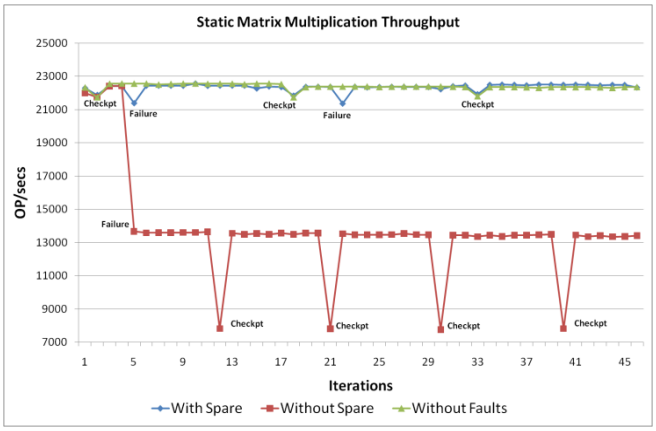


Figure 4. Throughput of the Matrix Multiplication application with and without spare nodes(32 processes – Checkpoint interval = 30 sec).

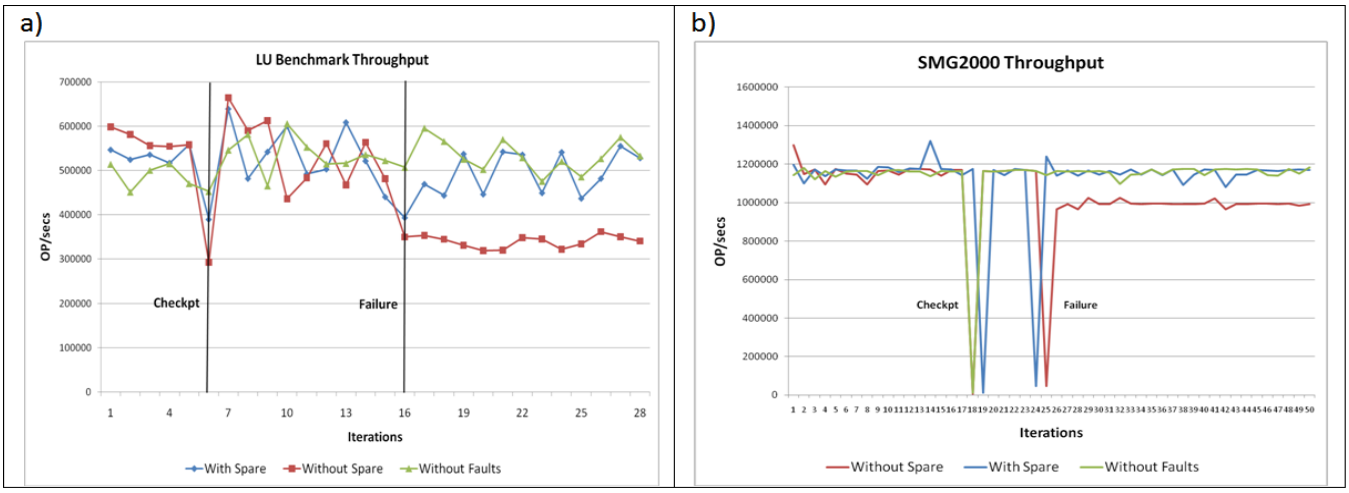


Figure 5. a) Performance of the LU Benchmark with and without Spare Nodes. b) Performance of the SMG2000 application with and without Spare Nodes.

performance degradation avoidance when failures occur in parallel computers. By using spare nodes automatically and transparently to restart failed processes into them, we can decrease the MTTR to a minimum while maintaining application performance as it was before failure.

As we mentioned before, it is crucial to deal with failures as fast as possible. If the application loses a node and we use the default approach of RADIC (Figure 2) one of the nodes become overloaded. As a consequence of this, the whole application throughput could decreases.

Replace failed nodes with spares is not trivial, because it is necessary to include the spare node into the parallel environment world and then restart the failed process or processes in it transparently and automatically. Therefore, application performance is affected only by a short period.

The experiments try to depict how performance (in terms of throughput) is affected after a failure without using spare nodes, and the benefits of using them.

To obtain the operations per second of the Matrix Multiplication application we divided the sub-matrix size that computes each process by the time spent into an internal iteration.

The checkpoint intervals that we use to make the experiments are only for test purposes. If we want to define valid checkpoint intervals we can use the model proposed in [19].

Figure 4 shows three executions of the matrix multiplication benchmark. The green line shows the fault-free execution. The blue line shows the execution of the application using RADIC with 2 spare nodes (and 2 failures). The red line shows the execution of RADIC without using spare nodes, and with one failure.

When a fault occurs and the application is not using spare nodes, the failed processes are restarted on their protectors and these nodes become overloaded. It occurs because processes compete for available resources and the application loses about 40% of its initial throughput. If we use RADIC with spare nodes, the application loses throughput only for a short period, until a spare is selected and the process is restarted in it, after that, the initial throughput is restored.

The static matrix multiplication is only a synthetic application. To evaluate the throughput changes with real applications using RADIC, we have designed a new set of experiments using the LU benchmark and the SMG2000 benchmark (Figure 5).

Figure 5a depicts the behavior of the LU benchmark using RADIC with and without spare nodes (and without faults). The application has an irregular behavior because it computes sparse matrices. If a failure occurs and the application does not have any spare node, it losses about 35% of its throughput. However, if the application uses spare nodes, the throughput is reduced in 25% during the recovery and after that, the initial throughput is recovered.

Figure 5b depicts the throughput of the SMG2000 benchmark. The checkpointing and restart operations are quite expensive for this application because the memory footprint of each process is about 2GB. If a fault occurs, and the process is restarted on its protector, the application losses about 15% of its initial throughput. However, if the process is restarted on a spare node, the initial throughput is maintained.

As is known, the execution time of an application affected by faults depends on the moment in which the failure occurs. For that reason when treating faults we focus on showing the degradation in terms of throughput not in terms of execution time.

Considering the results we can conclude that transparent and automatic management of spare nodes reduces avoids increments in the MTTR and maintains the application throughput avoiding system overload.

6 Conclusions and Future Work

The proposal presented in this paper has demonstrated that the RADIC policies are effective to automatically and transparently manage spare nodes, avoiding long recovery times while maintains initial application performance.

In this paper we have presented the design and evaluation of an alternative method to restart failed processes automatically maintaining the original computational capacity. This is an important issue because usually, applications are configured to execute with an optimal number of nodes. Loosing computational resources due to hardware failures decreases the application performance.

Having scalability as an objective, it is imperative to use a decentralized fault tolerance approach. Furthermore, when failures occur, a transparent an automatic fault treatment is desired, because the parallel application will experiment performance degradation only for a short period of time.

The use of a fault tolerance architecture with RADIC characteristics is desirable because it does not require any user intervention and it is also configurable to use available resources. When running parallel applications in computer clusters, frequently, there are free nodes that are not being used by any application, so these nodes could be used as spare nodes.

The implementation of RADIC into the Open MPI library has several advantages. The first one is that Open MPI is a widely used library used in the scientific world. It allows the utilization of RADIC with real scientific applications. The second advantage is that our implementation makes easier the processes migration without stopping the parallel application execution.

Initial analyses also show that RADIC will complement correctly with the MPI3 standard. The MPI3 standard will make easier the failure management because more information about failures will be available, so the possibilities to take corrective actions will increase.

The integration of RADIC into a stable Open MPI implementation as well as provide an interface for live migration are pending tasks. RADIC also need to start taking into account applications with I/O events (transactional applications).

7 References

[1] Bianca Schroeder and Garth A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, 2007.

[2] B. Randell, "System structure for software fault tolerance," *SIGPLAN Not.*, vol. 10, no. 6, pp. 437--449, April 1975.

[3] Amancio Duarte, Dolores Rexachs, and Emilio Luque, "Increasing the cluster availability using RADIC," *IEEE International Conference on Cluster Computing*, pp. 1--8, 2006.

[4] E. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375--408, September 2002.

[5] E Gabriel et al., "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," , 2004, p. 97-104.

[6] Guna Santos and Angelo Duarte, "Increasing the Performability of Computer Clusters Using RADIC II," *International Conference on Availability, Reliability and Security.*, pp. 653--658, 2008.

[7] Jason Duell, "The design and implementation of Berkeley Labs linux Checkpoint/Restart," 2003.

[8] Jason Ansel, Kapil Arya, and Gene Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," , 2009, pp. 1--12.

[9] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for Open MPI," *In Workshop on Dependable Parallel, Distributed and Network-Centric Systems(DPDNS), in conjunction with IPDPS*, pp. 1-8, 2007.

[10] Aurélien Bouteiller and Thomas Hérault, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *Supercomputing, 2003 ACM/IEEE Conference*, 2003, pp. 25 - 25.

[11] K M Chandy and Leslie Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63--75, February 1985.

[12] Darius Buntinas et al., "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols," *Future Generation Comp. Syst.*, vol. 24, pp. 73-84, 2006.

[13] Sriram Sankaran, Jeffrey M Squyres, Brian Barrett, and Andrew Lumsdaine, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," *In Proceedings, LACSI Symposium, Sante Fe*, pp. 479--493, 2003.

[14] Rinku Gupta et al., "CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems," *Parallel Processing, International Conference on*, vol. 0, pp. 237-245, 2009.

[15] Angelo Duarte, Dolores Rexachs, and Emilio Luque, "An Intelligent Management of Fault Tolerance in Cluster Using RADICMPI," pp. 150--157, 2006.

[16] Leonardo Fialho, Guna Santos, Angelo Duarte, Dolores Rexachs, and Emilio Luque, "Challenges and Issues of the Integration of RADIC into Open MPI," , 2009, pp. 73--83.

[17] D Bailey et al., "The Nas Parallel Benchmarks," *International Journal of High Performance Computing Applications*, 1994.

[18] Kiattisak Ngiamsoongnirn, Ekachai Juntasaro, Varangrat Juntasaro, and Putchong Uthayopas, "A Parallel Semi-Coarsening Multigrid Algorithm for Solving the Reynolds-Averaged Navier-Stokes Equations," , 2004, pp. 258--266.

[19] Leonardo Fialho, Dolores Rexachs, and Emilio Luque, "What is Missing in Current Checkpoint Interval Models?," *Distributed Computing Systems, International Conference on*, vol. 0, pp. 322-332, 2011.

A Distributed Service Architecture for Networked Automotive E/E System

Kabsu Han, Jeonghun Cho
School of E/E, Kyungpook National University, Daegu, Republic of Korea

Abstract - *Automotive Electric and Electronic systems come into wide use as requirement for more and complex functionality in vehicle is increase. Automotive E/E systems are based on networked ECUs, which each ECU is connected to each other through network, and distributed control system. As the ECUs are increase on the network, the development of automotive E/E system gets complicated. This paper considers the problems of automotive E/E system and proposes distributed service architecture, which can provide dependability and flexibility. Distributed service architecture separates service management from service processing, simplifies service layer and service node and makes development phase easy. Depend on service clustering. Proposed architecture separates services into sub-network from network and provides distributed service management.*

Keywords: In-Vehicle Network; Distributed Control Network; Fault-Tolerant Architecture; Dependable Distributed System;

1 Introduction

Electric, electronic and software are generally used in automotive system. Automotive electric/electronic system provides huge improvement of functionality, performance and product properties. But automotive industry does not have much experience in automotive E/E engineering; include software, embedded computer system and management. In fact, 49.2% of car break downs in Germany were due to E/E system failure in 2003 and also major recalls from automotive industry are based on problems of E/E system

One important point of automotive E/E system for automotive industry is the possibility to achieve competiveness through cost-efficient and dependable system. But development and modification of automotive E/E system is complicated and dangerous, even these are easier than mechanic/hydraulic system. This paper proposes distributed service architecture to solve these problems and to make automotive E/E system dependable.

Section 2 describes background for automotive E/E system. Section 3 describes the overview of distributed service architecture, section 4 presents service management, flexible implementation, fail-safety, on-line diagnosis and test. Section 5 describes experimental environment. Finally, the last section presents conclusion and future work.

2 Background

In early days, a vehicle has only a few Engine Control Units (ECU) and ECUs were not connected to each other and worked as stand-alone units, shown as fig. 1 a). It is simple enough to develop and modify, and barely need engineering efforts. Also, an ECU failure does not affect other ECUs.

As a development of electronics and embedded system technology, Electronic Control Unit (ECU) which is based on software, is used to implement automotive functionality. Automotive E/E system gets complex substantially to implement a lot of functions which demand for competitive strategy, customer requirement and law. The Mercedes S-Class included more than 50 ECUs from 1991 and the BMW 5 and 7 series have 70 networked ECUs from 2004.

In-Vehicle Network (IVN), which connects ECUs to ECUs, with point-to-point connections is the simplest topology and can make functionality easily, shown as fig. 1 b). But this topology has complex wiring which is hard to modify and extend. And as ECUs are increase, the wiring and connection points are increase exponentially. This is not good for development, production and maintenance of vehicle. The Mercedes S-class from 1991 has more than 3km of wiring.

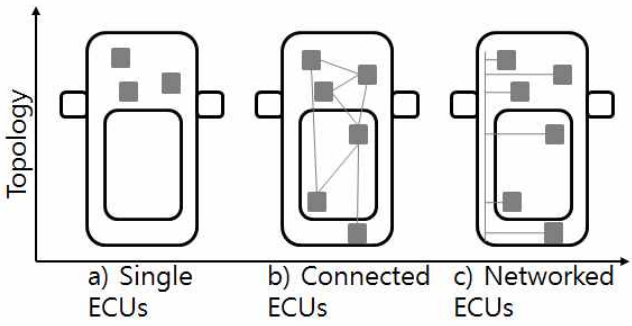


Figure 1 Automotive E/E architecture

Networked E/E system uses bus architecture for simplicity to develop, shown as Fig. 1. c), also flexibility to modify easily. ECUs, which have their own sensors and actuators, are

connected with others through network and are able to share some information from sensor. Although networked E/E system has some advantages compared to point-to-point connection, still some limitation are remaining to provide flexibility and scalability. Sensors, actuators and other properties do not have their own network interface, so they have to be a part of some kind of ECUs.

Fully networked E/E system consists of smart sensors and smart actuators which have their own network interfaces thus they can connect to IVN independently and assure high flexibility and scalability. But with the increase in number of ECUs, sensors and actuators, the development of Automotive E/E system becomes complicated. Moreover, heavy traffic on IVN causes congestion, delay and emission of data, thus safety and dependability are not sure. This paper proposes distributed service architecture to solve problems, which are described above, and provide flexibility. Distributed service management points (SMP) maintain services hierarchically and provide redundant services to assure dependability. According to the requirements, SMP can consist of various services and can modify services easily and support on-line diagnosis and test to validate services.

3 Basic System Architecture

Proposed distributed service architecture consists of a network of SMPs and Service Control Point (SCP). The functionality of SMP is a service management of system with service state messages and relays the service requests. Each SMP is connected with other SMPs through network but SMPs do not have any information over the other SMPs, shown as fig. 2. With service index table, SMP can only lookup available services and SMPs which manages services.

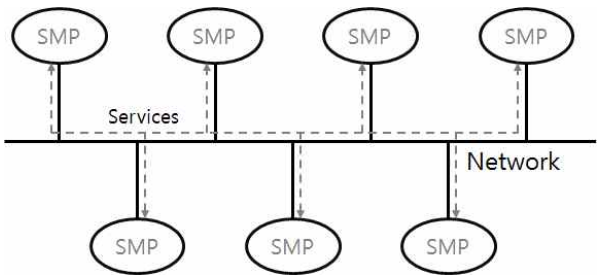


Figure 2 Simple view of SMPs

SMP manages service states of SCPs and relays service request from SCP to specific SMP which can process requested service. Each SMP checks state of other SMPs with service state message, maintains service index table which describes service information. Each SMP has its SCPs which are connected to sub-network. Depend on functional and strategic requirements of automotive E/E system, the composition of SMPs and SCPs, which called service clustering, can vary considerably. In most cases, service cluster and services are defined in system analysis documents already.

If service is processed within a SMP, it called in-bound service. If service is not processed within a SMP, SMP has to relay service request to other SMPs, it called out-bound service, shown as fig. 3. In case of in-bound service, requested services do not make any data on main network, only on sub-network which separated from main network.

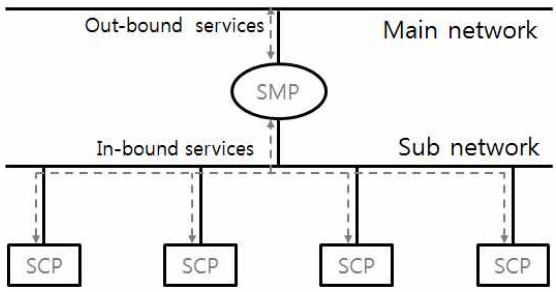


Figure 3 Basic composition of SMP and SCPs

SCP consists of ECUs, sensors, actuators and other properties. SCP processes relayed service request from SMP with its own resources. When some kind of event happened, SCP sends service request to SMP simply and processes relayed service request from SMP. If SCP needs some service it sends a service request to the SMP to which it is directly connected. If the requested service is available within the same SMP, in-bound service, it is relayed to SCP, which can process the service, on the sub network. On the other hand, if the service is unavailable, out-bound service, the SMP relays service request to another SMP on main network. In case of no SMP on the network is able to process the service, the SMP replies with a service fail message. Also, service requests and service replies are define in system analysis documents. The entire architecture is shown as fig. 4.

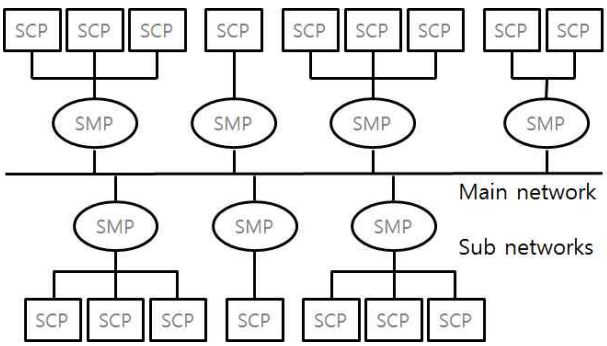


Figure 4 Overview of Distributed Service Architecture

Logging SMP can collect all of service request, service state and service failed messages. Logging SMP is connected with SCPs which are related to log services and provide on-line diagnosis with various logging data services.

Depend on service clustering, proposed distributed service architecture separates sub-network from main network. The separation of services can reduce numbers of connection points and disperse network traffic. Comparing state messages of SMPs, the proposed architecture can detect faults of SMPs and can operate in fail-operational mode with redundant SMPs. Additionally, SMPs can isolate fault of SCPs and redundant SCPs can cover faults of other SCPs.

4 Distributed Service Architecture

This section describes service management of proposed architecture and flexible configuration in implementation level. Also, describe reliability and construction of on-line diagnosis and test.

4.1 Service Management

Distributed service architecture adopts point-to-point service architecture in general distributed systems. Proposed architecture manages services and cooperates without central supervisor and central arbiter [1][9]. Because of single point failure problem, complexity and flexibility, central control supervisor is not good for distributed control system. Automotive E/E system is a kind of real-time control system which is based on a set of control units with software and interfaces interconnected on network. Depend on system requirement, allocation of control units on the network will be various as well as allocation of interfaces and software on the control units. But services have to be finished within deadline, which defined in the system requirement, although system has different design. We assume that proposed architecture just has simple layers, which consist of single SMP layer and single SCP layer, to assure service result within deadline and to provide simplicity for flexible reconfiguration. To manage services on the distributed service architecture, each SMP has 3 information tables, service index table, service page table and service list table.

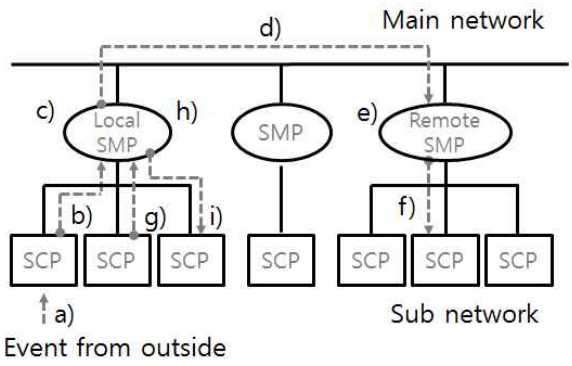


Figure 5 Basic service flow

In proposed architecture, basic service flow is shown as fig. 5. SCP detects events from outside, e.g., data from speed sensor or ACC on/off switch, and sends service request to local SMP which is connected directly with SCP. When local SMP receives service request from SCP, local SMP lookups service in service index table and relays the service to remote SMP, in case of out-bound service, or SCP, in case of in-bound service. If local SMP does not find service in service index table, local SMP replies to SCP with service fail message. When remote SMP receive the service request, remote SMP lookup the service in service index table, relays the service to SCP. Also, if SCP is not available, remote SMP has to reply with service fail message. If service is available in Local SMP, local SMP relay the service to SCP.

Service index table, which has service ID and information of SCPs and SMPs, is defined statically in system analysis documents. Each SMP maintains information of service index table dynamically with exchange of service state messages. Service index table, shown as Table 1, consists of service list ID, service page ID and service flag. SMPs have to share same information of service index table to assure service state. Service list ID identifies service uniquely. Service page ID represents a group of SMP which manages specific services. Service flag indicates service state, if service is available on a SMP at least, service flag is true. If not available on all of SMPs, service flag is False.

Table 1 Service Index Table

Service List ID	Service Page ID	Service Flag
List ID 1	Page ID 1	T
List ID 2	Page ID 1	T
List ID 3	Page ID 1	T
List ID 9	Page ID 2	T
List ID 10	Page ID 2	T
List ID 11	Page ID 9	F
…	…	…

According to service clustering and design policy, service page table is defined statically, shown as Table 2. SMPs have to share same information of service page table, also. For dependability and simplicity, service page table separate from service index table. Because service index table indicates service state only, SMP can recognize service state easily without lookup of all of SMPs. if service needs to be moved to another SMP, service page table will be modified only.

Service list table has information of SCPs which correspond to services, shown as Table 3. Each SMP has its own service list table, it is local information.

Table 2 Service Page Table

Page ID	Primary SMP	Secondary SMP
Page ID 1	SMP1	SMP4
Page ID 2	SMP2	SMP5
Page ID 9	SMP3	SMP2
…	…	…

Table 3 Service List Table

List ID	Primary SCP	Secondary
List ID 1	SCP1	SCP4
List ID 2	SCP2	SCP5
List ID 3	SCP3	SCP1
…	…	…

When local SMP receives service request from SCP, local SMP finds service list ID of service request in service index table. If local SMP finds service list ID in service index table, it retrieves service page ID. Local SMP finds information of SMP in service page table with service page ID, shown as fig. 6. In case of in-bound service, local SMP finds information of SCP in service list table with service list ID. In case of out-bound service, local SMP relays service request to remote SMP. When remote SMP receives service request from local SMP, remote SMP finds information of SCP in service list table with service list ID, also. In case of both, SMP can find SCPs in service list table and relays service request to SCP.

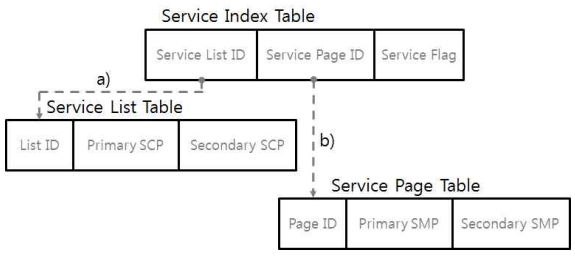


Figure 6 Relation of service tables

Table 1 is example of service index table, 6 services and 3 pages are in service index table. Service List ID 1, 2 and 3 belong to Page ID 1 and all of these services are available. Service List ID 9 and 10 belong to Page ID 2, service is available, also. Service List ID 11 belongs to Page ID 9, service is not available. When SMP receives service request with List ID 9, it retrieves Page ID 2 from service page table. Page ID 2 consists of SMP 2 and SMP 5 in service page table. From Table 2, Primary SMP is SMP 2, SMP will relay service List ID 9 to SMP 2.

Table 3 is an example of service list table of SMP1. When SMP want to relay service List ID 2 to SCP, SMP just retrieves List ID 2 in service list table. List ID 2 has two SCPs, SCP 2 for primary SCP and SCP 5 for secondary SCP in service list table. SMP will relay service list ID 2 to SCP2.

Each SMP checks service state of SCPs periodically and manages service flag in service index table. Also, if an event occurs, SMP updates service flags and notify immediately. When SMP detects that service state of SCP is changed, SMP update service state and sends service state message to other SMPs..

4.2 Flexible Implementation

Proposed distributed service architecture provides flexible implementation. Depending on system requirement, main network and sub-network can be implemented with various digital control networks [2]. Generally, CAN, FlexRay, and MOST which have high bandwidth and dependability, are used for main network. Especially, FlexRay and FT-CAN, which have redundant communication channel, are good for safety and dependability but not for flexibility. Sub network often uses CAN and LIN, which are cheap and easy to use. SMP and SCP can construct various hardware and software architecture; also, SMP and SCP can be on same hardware (like a multi-processor, SoC) in implementation level. SCP can be implemented as various combinations of ECUs and sensors, actuators, depending on system requirement and restriction.

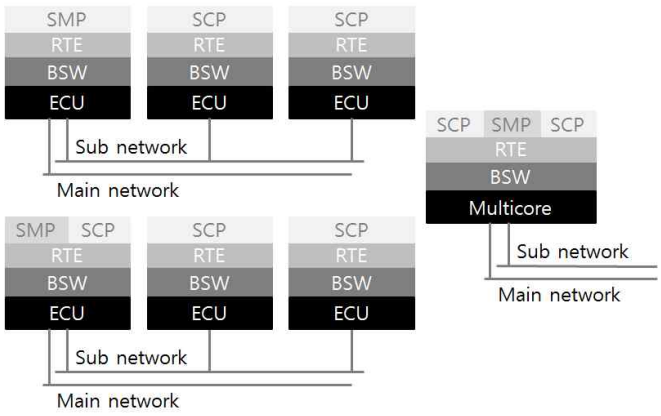


Figure 7 Flexible Implementation

Proposed architecture separates sub-network from main network. Because of this separation, numbers of ECUs, sensors and actuators are decreased on main network and network traffic is dispersed. Also, design verification and validation of automotive E/E system will be easier. Physical

restriction of network can be avoided, that's why the physical link of network can be shortened. Traditionally, IVN consists of power-train, chassis, body and entertainment domains, each domain works independently. But, they need to be connected with each other to share information, e.g., Adaptive Cruse Control needs information and control of power-train and chassis domain. Infotainment system needs information of all of domains.

Depend on system requirements, SMPs may support various network and sub-network in implementation level, shown as Table 4. For example, CAN 2.0, which generally used in automotive E/E system, for main network, CAN 1.0 and LIN for sub-network are possible. Also, FlexRay, which based on time-triggered protocol, for main network, CAN and LIN for sub-network are possible.

Table 4 Example Composition of Network

	Main Network	Sub-Network
Example 1	CAN 2.0	CAN 1.0, LIN
Example 2	FlexRay	CAN, LIN
Example 3	MOST	CAN, LIN

In case of CAN network, CAN use priority-based protocol, a kind of event-triggered, for multiple access control. To implement functions, priority assignment methods for CAN messages are needed in design phase. The design of priority assignment gets more and more complicated as the number of connection points, numbers of messages are increase. Low priority messages cannot be transferred through the network because of jitter and delay introduced after the implementation. To solve this problem, flexible priority of CAN messages are proposed [5], but this is not good for safety and reliability of the entire system. Moreover, as physical link runs long, transfer rate is going down, e.g., maximum transfer rate is up to 1Mbps at 40m but 250Kbps at 200m and 50Kbps at 1km.

In case of FlexRay, FlexRay use TTP (Time Triggered Protocol) for multiple access control, design of slot allocation of static segment is needed. According to static slot allocation, messages are transferred in each static segment. The rest of a time segment, message is transferred as a dynamic segment. Similar to CAN, as numbers of connection-points are increase, the design of slot allocation gets complicated exponentially. 74 parameters need to be configured with suitable value within predefined range. Possible configuration spans more than 10^{48} for each design. Lack of flexibility and sub-optimal resource usage problem are still remaining [13]. In worst case, messages cannot be allocated in a static segment because the numbers of slots in a static segment are limited, e.g., 16 bytes payload, 3ms static segment cycle will be 93 slots at 10Mbps, 51 slots at 5Mbps and 27 slots at 2.5Mbps [14].

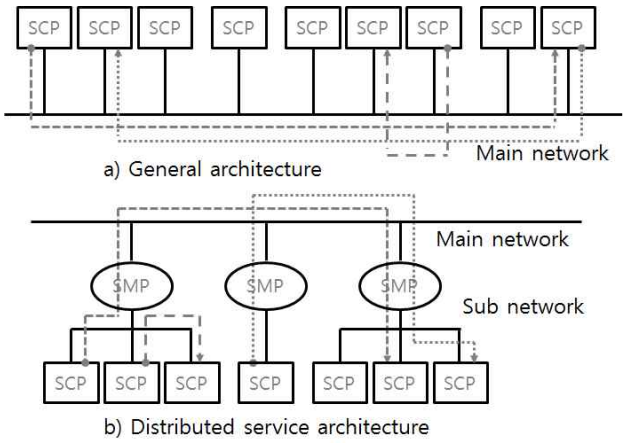


Figure 8 Comparison of service flow

The Proposed architecture separates sub-network from main network, only SMPs are connected to main network, the number of connection-points will be decrease, shown as fig.8. As only out-bound services are transferred through the main network, network traffic will be decrease and design, validation and verification will be easier. Furthermore, as physical link is shortened, loss of transfer rate can be minimized.

4.3 Fail Safety

A fault-tolerant system has redundant components to assure dependability and safety generally [3][4][8]. The proposed architecture provides distributed fault tolerant architecture. Redundant SMP in service page table and redundant SCP in service list table minimize the failure of the services. To detect software faults, at least, two version of software are needed. Also, to detect hardware faults, more than two types of hardware are needed. Use of different version of software can detect software fault and hardware transient fault, shown as Table 5. And use of different types of hardware can detect hardware permanent fault.

Table 5 Fault detection case

S/W (# of version)	H/W (# of hardware)	Fault Type		
		S/W	H/W Transient	H/W Permanent
1	1	X	O	X
1	2	X	O	O
2	1	O	O	X
2	2	O	O	O

Each SMP can manage primary SCP and secondary SCP in service list table for fault-tolerant system as well as SMP in service page table. Different software on SCP, for fault detection, is available. Depending on system requirements for safety and dependability, fault-tolerant SCP runs in dual mode. In first mode, all of SCPs can fully process services. When run in fail-operational mode, secondary SCP processes only degraded services.

4.4 On-line Diagnosis and Test

Logging SMP can collects all of state messages and service messages while logging SCPs process those messages on development phase and product phase. On-board computer, which can process simple diagnosis, can be a kind of logging SCP. Off-board interfaces, like OBD-II, for intensive diagnosis, can be a kind of logging SCP, shown as fig. 9[6].

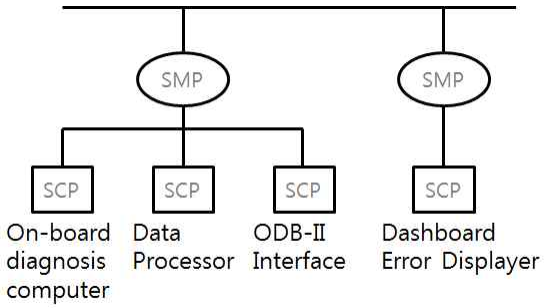


Figure 9 Example of on-line diagnosis

Collection and analysis, statistics, data-mining of messages are processed by logging SCPs. As the composition of logging services, messages can be processed efficiently and systematically for a long time during development phase. Simple logging SCPs can be used du 0.05 cming production phase for maintenance. As the increase of services, various SCPs can be used for on-line diagnosis. Flexible on-line diagnosis can be possible for domain characteristics of vehicle and requirement.

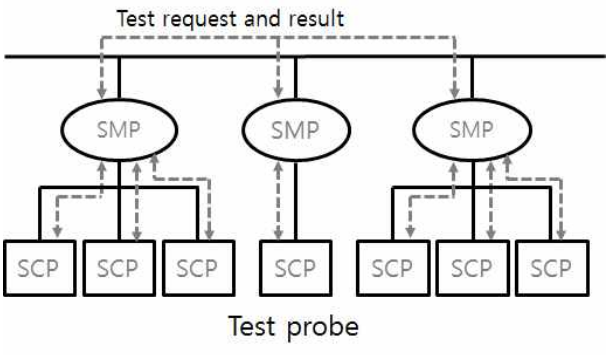


Figure 10 Example of test probe

For test automation environment, proposed architecture can configure SCPs as test probes, shown as fig. 10. To verify and validate service list and service configuration, test probe sends all of service request automatically as an acceptance test. If service list and service configuration are not correct, service fail message will be returned. Input of test cases, e.g., service ID, service table configurations, and output, e.g., expecting service result, can be generated automatically from system design documents. When service configuration is changed for optional service and variation, test automation with test probes and test case is good for verification and validation.

5 Experimental Results

5.1 Experimental environment

Experimental environment, adaptive front lamp control system, consists of three SMPs and four SCPs, shown as fig. 11. Assume that two SCPs in drive shaft cluster detect degree of steering wheel and speed of vehicle and send control request to SMP. SCP, which detects degree of steering wheel, sends control requests of yaw of the front lamps. SCP, which detects speed, sends information of control speed. As the speed of vehicle, front lamps have to move fast or slow. SMP with two SCPs use CAN for a sub-network, because of sampling rate of data. SMP with yaw SCPs use LIN as a sub-network. test SMP and test SCP is on same hardware and CAN use for main network.

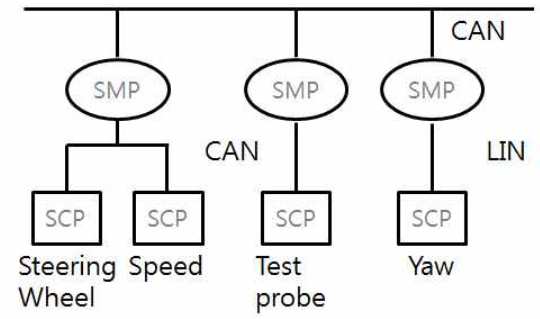


Figure 11 Experimental Environment

Freescale MC9S12XF512 board is used for SMPs which have two interfaces of each LIN, CAN, Flexray. MC9S12XF512 board is working in stand-alone mode, shown as fig. 12. Software of SMP is developed with Freescale Cordwarrior IDE and BDM interface.

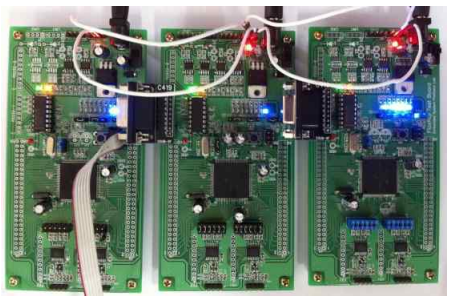


Figure 12 Implementation of SMPs

5.2 Experimental Scenarios

We design service list and service configuration of system. To manage service state, simple service flag message is developed. When SMP detects service change of SCPs, SMP sends service flag message to others. Only service flag message can change service state of service index table. If there is no corruption, all SMPs have same service state.

Service message consist of service request message and service fail message. It is possible that local SMP sends service request message to remote SMP before local SMP change service state with service flag message. In this case, remote SMP will reply with service fail message. When local SMP receives service fail message, Local SMP retrieve service state and try again. Out-bound service can be validated with CAN analyzer, shown as Fig.13.

Protocol	Arbiter	Length	DATA(HEX)	DATA(ASCII)
Std(2.0A)	DA E0	8	53 4D 50 33 53 54 41 ...	SMP3STAT
Std(2.0A)	ED E0	7	53 76 63 35 52 65 71	Svc5Req
Std(2.0A)	EA E0	7	53 76 63 33 52 65 71	Svc3Req
Std(2.0A)	E9 E0	7	53 76 63 32 52 65 71	Svc2Req
Std(2.0A)	DA E0	8	53 4D 50 33 53 54 41 ...	SMP3STAT
Std(2.0A)	D9 E0	8	53 4D 50 32 53 54 41 ...	SMP2STAT
Std(2.0A)	D8 E0	8	53 4D 50 31 53 54 41 ...	SMP1STAT
Std(2.0A)	ED E0	7	53 76 63 35 52 65 71	Svc5Req
Std(2.0A)	EB E0	7	53 76 63 34 52 65 71	Svc4Req
Std(2.0A)	E8 E0	7	53 76 63 31 52 65 71	Svc1Req
Std(2.0A)	DA E0	8	53 4D 50 33 53 54 41 ...	SMP3STAT
Std(2.0A)	EA E0	7	53 76 63 33 52 65 71	Svc3Req
Std(2.0A)	ED E0	7	53 76 63 35 52 65 71	Svc5Req
Std(2.0A)	E9 E0	7	53 76 63 32 52 65 71	Svc2Req
Std(2.0A)	D8 E0	8	53 4D 50 31 53 54 41 ...	SMP1STAT
Std(2.0A)	D9 E0	8	53 4D 50 32 53 54 41 ...	SMP2STAT
Std(2.0A)	DA E0	8	53 4D 50 33 53 54 41 ...	SMP3STAT
Std(2.0A)	E8 E0	7	53 76 63 31 52 65 71	Svc1Req
Std(2.0A)	ED E0	7	53 76 63 35 52 65 71	Svc5Req
Std(2.0A)	EB E0	7	53 76 63 34 52 65 71	Svc4Req
Std(2.0A)	EA E0	7	53 76 63 33 52 65 71	Svc3Req
Std(2.0A)	DA E0	8	53 4D 50 33 53 54 41 ...	SMP3STAT
Std(2.0A)	E9 E0	7	53 76 63 32 52 65 71	Svc2Req
Std(2.0A)	D8 E0	8	53 4D 50 31 53 54 41 ...	SMP1STAT
Std(2.0A)	ED E0	7	53 76 63 35 52 65 71	Svc5Req
Std(2.0A)	D9 E0	8	53 4D 50 32 53 54 41 ...	SMP2STAT
Std(2.0A)	E8 E0	7	53 76 63 31 52 65 71	Svc1Req
Std(2.0A)	DA E0	8	53 4D 50 33 53 54 41 ...	SMP3STAT
Std(2.0A)	EB E0	7	53 76 63 34 52 65 71	Svc4Req
Std(2.0A)	EA E0	7	53 76 63 33 52 65 71	Svc3Req
Std(2.0A)	ED E0	7	53 76 63 35 52 65 71	Svc5Req
Std(2.0A)	D8 E0	8	53 4D 50 31 53 54 41 ...	SMP1STAT
Std(2.0A)	DA E0	8	53 4D 50 33 53 54 41 ...	SMP3STAT
Std(2.0A)	E9 E0	7	53 76 63 32 52 65 71	Svc2Req
Std(2.0A)	ED E0	7	53 76 63 35 52 65 71	Svc5Req
Std(2.0A)	E8 E0	7	53 76 63 31 52 65 71	Svc1Req

Figure 13 CAN Messages Monitoring

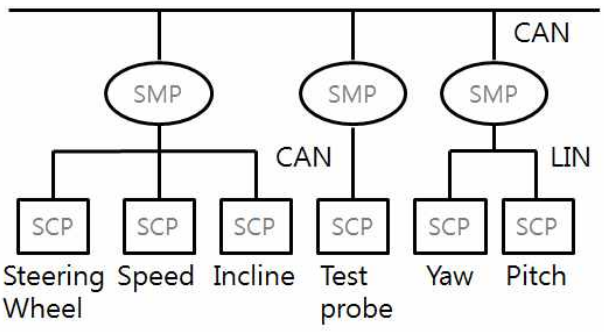


Figure 14 Reconfigured experimental environment

Now, assume that new service is added that is known as front lamp auto leveling. For this service, two SCPs, which can detect incline and control pitch of the front lamps, will be added to experimental system, shown as fig. 14. Incline SCP, pitch SCP are developed, also. SCP, which detects incline of vehicle, sends control request of pitch of front lamps.

For service reconfiguration, just add service list id to service index table, SMP to service page table and SCP to service list table. Services are verified and validated with test probe. Service list is allocated to CAN message, shown as Table 6.

Table 6 Design of messages

Message type	CAN ID
Service Flag	0xD8E00000
Service Notification	0xD9E00000
Service Interrogation	0xDAE00000
Service Fail	0xE8E00000
Service Req1(Left)	0xE9E00000
Service Req2(Right)	0xEAE00000
Service Req3(Up)	0xEBE00000
Service Req4(Down)	0xECE00000
Service Req5(Reset)	0xEDE00000

6 Conclusions

This paper proposed distributed service management to solve complexity of traditional in-vehicle network architecture and provide flexibility. The proposed architecture can use various control networks in implementation level. Because of separation of service management from processing, distributed SMPs manage their own SCPs simply and just relay service requests. As the sub-network is separated from main network, design, verification and validation of vehicle network is easier than before. Redundant SMPs and SCPs improve safety and dependability of system. Proposed architecture can be used other distributed control area. For design and implementation of automotive E/E system, EAST-ADL (Architecture Description Language) and Automotive Open system Architecture(AUTOSAR) can be used [10][11][12].

Research about the framework to define and design services and system is needed. Requirement analysis, system configuration and formalized development process, which are suitable for de-facto of the automotive industry, will be added to our future work. Moreover, research on implementation platform for SCPs and SMPs with AUTOSAR complaint is also needed. Finally, Test interfaces and test platform for verification and validation are needed for safety and dependability, also.

7 ACKNOWLEDGEMENTS

This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the CITRC(Convergence Information Technology Research Center) support program (NIPA-2012-C6150-1202-0011) supervised by the NIPA(National IT Industry Promotion Agency) and the Ministry of Education, Science Technology (MEST) and National Research Foundation of Korea(NRF) through the Human Resource Training Project for Regional Innovation.

8 References

[1] H. Kopez, R. Obermaisser, C. El Salloum, B. Huber, "Automotive Software Development for a multi-core System-on-a-chip," in ICSE workshops SEAS'07, Minneapolis, 2007.

[2] C. Pinello, L. P. Carloni, A. L. sangiovanni-Vincentelli, "Fault-Tolerant Deployment of Embedded Software for Cost-Sensitive Real-Time Feedback-Control Applications," IEEE Design, Automation and Test in Europe Conference and exhibition, vol. 2, pp. 1164-1169, Febrary 2004.

[3] J.R. Pimentel, M. Salazar, "Dependability of Distributed Control system Fault Tolerant Units," in IECON 02, vol. 4, pp. 3164-3169, November 2002.

[4] Yeon Kyu Bong, Jeong Ki Yun, "The Dependability Analysis of LIN Network for Adaptive Front-Lighting System," in ISOCC'08, vol. 1, pp.425-428, November 2008.

[5] Tan Xiao-peng, Li Xiao-bing, Xiao Ti-Liang, "Real-time Analysis of Dynamic Priority of CAN Bus Protocol," in ICEIE 2010, vol. 1, pp. 219-222, August 2010.

[6] H. Schweppe, A. Zimmermann, D. Grilly, "Flexible In-Vehicle Stream Processing with Distributed Automotive Control Unis for Engineering and Diagnosis," in SIES' 08, vol. 1, pp. 74-81, June 2008.

[7] J. Schäuffele, T. Zurawka, R. Carey, "Automotive Software Engineering", SAE International, 2005

[8] S. Pledna, H. Kopez, "Fault-Tolerant Real-Time Systems", Kluwer Academic Publishers, 1996

[9] Z-M Gu, D-G Yang, X-F Zhang, L LU, K-Q Li, X-M Lian, "Distributed Vehicle Body electric/electronic system architecture with central coordination control," IMechE 2010, vol. 224, Part D, pp.189-199, Febrary 2010.

[10] C. Pfaller, A. Fleischmann, J. Hartmann, M. Rappl, S. Rittmann, D. Wild, "On the Integation of Design and Test – A Model-Based Approach for Embedded Systems," in AST'06, Shanghai, vol. 1, pp. 15-21, May 2006

[11] U. Freund, V. Jaikamal, J. Löchner, "Multi-level System Integration of Automotive ECUs based on AUTOSAR," SAE World Congress & Exhibition 2009, Detroit, 2009

[12] M. Rohdin, L. Ljungberg, U. Eklund, "A Method for Model Based Automotive Software Development," ARTES, 2000

[13] E. Armengaud, A. Steininger, M. Horauer, "Automatic Parameter Identification in Flexray based on Automotive Communication Networks", in ETFA'06, vol.1, pp897-904, September, 2006

[14] M. Grenier, L. Havet, N. Navet, "Configuring the communication on Flexray – the case of the static segement", in ERTS'08, January, 2008

A File System Using GPU-Accelerated File-wise Reliability Scheme

Chien-Kai Tseng, Shang-Chieh Lin, and Yarsun Hsu
Department of Electrical Engineering, National Tsing Hua University, HsinChu, Taiwan, R.O.C

Abstract— *This work revises the original file-wise reliability scheme to cope with larger pages in storage devices nowadays, and implements it as a file system prototype: CRSFS. There are four layers in CRSFS: GPU primitive for Cauchy Reed-Solomon (CRS) coding, CrystalGPU framework, CRS coding layer and AFS FUSE layer.*

CRSFS provides GPU acceleration on the CRS encoding/ decoding operations by using the CUDA program: GPU primitive for CRS coding. Besides, it is integrated with FUSE (Filesystem in Userspace) framework and Rx (extended remote procedure call) protocol in AFS FUSE layer to provide high flexibility on storage system configurations. Hence, most programs can benefit from it without rewriting their read/ write operations.

Finally, it's shown that with the help of GPU acceleration, there are up to 24.5x performance gains compared to the CPU counterpart in AFS FUSE layer. The speed of CRS encoding/decoding operations is no longer the performance bottleneck.

Keywords: Reliability; file-wise reliability; Fault-tolerance; File System; CUDA; GPU acceleration

1. Introduction

Reliability is always a big concern for storage devices. Nearly all the storage devices have some kind of mechanism to ensure the correctness of the data stored on them.

As for large storage systems, the Redundant Arrays of Inexpensive Disks (RAID) [1] technique is commonly applied to prevent data losses caused by disks failures. However, because of the high cost and low flexibility of the RAID configuration on storage devices, the file-wise reliability scheme has been proposed to provide different level of protections based on the importance of files [2]. But the Reed-Solomon (RS) code encoding/decoding operations are time-consuming when executed on CPU. Moreover, the erasure unit size proposed at that time is no longer suitable for the storage devices nowadays. The page size in Solid-State Drives (SSDs) has increased to 4 KB or 8 KB. Even the sector size in hard disks has increased from 512 bytes to 4 KB. Therefore, it is necessary to redesign the file-wise reliability scheme and introduce the GPU acceleration to improve the encoding/decoding speed.

In this paper, the original file-wise reliability scheme is revised to deal with larger page sizes in current storage devices, and implemented in the file system prototype — CRSFS. In the revised file-wise reliability scheme, there are total four different reliability levels. All of them utilize CRS code to satisfy different file reliability requirements between ordinary user files and important system files.

Due to the low throughput of RS encoding/decoding operations on previous work, CRSFS is equipped with GPU acceleration to improve the speed of CRS encoding/decoding operations. Besides, CRSFS integrates FUSE [3] and Rx [4] to provide high flexibility on storage system configurations.

The introduced CUDA program in this work—GPU primitive for CRS coding—is optimized for the Fermi architecture of Nvidia GPUs and is integrated with CrystalGPU [5] to exploit the ability of concurrent computation and communication and eliminate the overhead of device memory allocations. In this work, it's shown that with the help of GPU acceleration, the performance gain can reach up to 24.5x compared to the CPU counterpart. Therefore, the speed of encoding/decoding operations is no longer the performance bottleneck.

The POSIX extended attribute APIs of file system are used in this work. Hence, any programs can manipulate the reliability level of a file by using setxattr(), getxattr() and removexattr() to change the *user.crsfs.rlevel* value of the file.

The rest of the paper is organized as follows: Section 2 introduces related work. Section 3 explains the design and implementation thoroughly. The performance evaluations and comparisons between GPU and CPU are presented in section 4. Finally, section 5 describes conclusions.

2. Related Work

The GPGPU computing is quite popular in recent years. Since then, researchers have exploited the GPU power to accelerate many real-world applications, including reliability mechanisms applied on storage devices. For example, a GPU-based RAID system is proposed as a user space framework to accelerate the RS code operations via table lookups [6]. Later, this system is integrated into Gibraltar RAID [7], which is executed on a server node to provide software RAID encoding/decoding services via high-speed network connections.

Other works, such as the Barracuda micro driver architecture for GPUs [8], try to leverage the GPU power in kernel space. Barracuda uses the CUDA based Peter Anvin Reed Solomon encoding (PARSE) [9] to provide the software RAID service to the file requests in kernel space. Due to the fact that the CUDA APIs are only available in user space, the data must be forwarded from kernel space to user space before processed by the CUDA program.

Gibraltar RAID provides services purely in user space, while the Barracuda provides services to the file requests in kernel space via procfs signaling forwarding. Both works are based on RS coding and use the table lookup technique to handle the complex arithmetic operations over Galois field (GF). Besides, they also provide the same software RAID service which can only handle reliability issues based on storage devices, rather than files.

In addition, the GPU accelerated storage system, which consists of MosaStore storage system, HashGPU library and CrystalGPU, is proposed as a content addressable storage system to provide hash service [10]. In that work, CrystalGPU enables up to 3x performance gains with single-GPU configuration compared with HashGPU, which is shown to have up to 5x performance gains compared with CPU.

Also, GPGPU has been shown to be a powerful off-loading engine for CRS encoding in [11]. All of these works above show that GPU has great potential of improving the parity generation performance on CPU.

3. Design and Implementation

In this section, the file-wise reliability scheme is reintroduced with some modifications. And the implemented file system prototype (*CRSFS*) is composed of four layers: GPU primitive for CRS coding, CrystalGPU framework, CRS coding layer and AFS FUSE layer.

3.1 Revised File-wise Reliability Scheme

We've proposed a file-wise reliability scheme in 2010 [2]. The idea of file-wise reliability scheme is allowing users to configure different levels of reliability on individual files according to the respective requirement, and is realized by attaching new abilities in storage subsystem of OS. These abilities are listed below:

- Record the reliability level of each file and store the checksum transparently
- Let users configure the reliability level of demanded files
- Automatically perform possible fault-tolerance when encountering I/O errors

The configuration of the revised file-wise reliability scheme is described in table 1. The CRS code for erasure coding is used to provide erasure recovery in all reliability levels. It takes 512 KB as data input of each encoding, and generates 32-KB, 64-KB, 128-KB and 256-KB redundancy in level 1, 2, 3, and 4 respectively. The larger erasure unit size(4KB) can help deal with large page size in SSDs and the Advanced Format sectors in HDDs.

Configuration or change to the reliability level of a file is performed via the POSIX extended file attribute system calls. Any change to the value of the extended file attribute named "*user.crsfs.rlevel*" invokes the encoding routine. And the recovery procedure is triggered transparently when encountering failed fetching of data pages, i.e. read errors. All these operations will be explained thoroughly in section 3.5.

Table 1: Algorithms and fault-tolerance capability (512 KB) in revised file-wise reliability scheme.

Level	1	2	3	4
Algorithm	(128,8)CRS	(128,16)CRS	(128,32)CRS	(128,64)CRS
Erasure Recovery	8 pages	16 pages	32 pages	64 pages

(k, m)CRS denotes Cauchy Reed-Solomon code with k data units and m checksum units. Packet size equals 512 bytes.

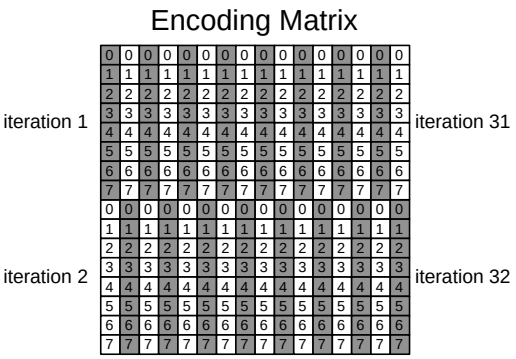


Fig. 1: Scheduling of CUDA threads according to the encoding matrix

3.2 GPU Primitive for CRS Coding

The CRS code operation for erasure coding is the matrix multiplication over $GF(2^1)$, thus converting all the operations over $GF(2^\omega)$ into bitwise *xor* and bitwise *and* operations. As a result, each column of the encoding matrix uses the same row of the data, and each row of the coding matrix contributes to the same row of the checksum.

The overall encoding computation is a $M^{m\times1024} \times N^{1024\times4096}$ matrix multiplications, and m equals 64, 128, 256 or 512. To apply the tile-based algorithm and maximize the parallelism, the workload is divided both by the rows of the encoding matrix and by the columns of the data chunk respectively. Therefore, each two-dimension block consists of $256(32 \times 8)$ threads, and is responsible for $M^{16\times1024} \times N^{1024\times1024}$ matrix multiplications.

At startup, the first warp in every block is responsible to load the encoding matrix from global memory to shared memory. Then, in each iteration, threads in the same block load one $matrix^{16\times32}$ block and one $data^{1024\times32}$ block by applying the *coalescing* technique to access continuous global address. Each element in $matrix^{16\times32}$ and $data^{1024\times32}$ is 32-bit wide. There are total $\frac{BLOCKROW}{THREADROW}(\frac{16}{8} = 2) \times BLOCKCOL(32) = 64$ iterations in the computation stage, starting from the upper-left corner of the encoding matrix along side with the same

column down to the lower-right corner as illustrated in fig. 1. In fig. 1, the elements of the encoding matrix filled with the same color are examined at the same iteration in the same CUDA block, and the number in each box represents the idy of the threads reading this value, where the idy represents the y dimension index. Every CUDA block will go through $M^{16\times1024}$ in this order to compute $C^{16\times1024}$.

In each computation iteration x_i, threads with the same idy check if the $\lfloor\frac{x_i}{2}\rfloor$-th element in the idy-th row of the the encoding matrix, denoted as matrix_val, is equal to one. If so, each of the threads does one 32-bit XOR operation, resulting in one large 1024-bit XOR operation of one $data^{1\times1024}$ block and one $cs^{1\times1024}$ block. After that, the result block is then written back to the same $cs^{1\times1024}$ block in shared memory. If matrix_val is equal to 0, then, they just skip current computation iteration. Because the threads with the same idy are in the same warp and they do the XOR operation according to the same matrix_val, there's no branch divergence.

The CRS GPU primitive is modified from an open source project - MAGMA [12] to take advantage of the Fermi architecture. The techniques used in MAGMA, such as texture cache and software pipeline, improve the matrix multiplication performance.

3.3 CrystalGPU Framework - revised

The CrystalGPU framework has been modified to better fit the need of CRS coding layer.

First, the rlevel is added to the data structure, gpufw_job_s, to record the reliability configuration of the current operation.

Second, gpufw_job_put_free() has been abandoned. Instead, the GPU master thread automatically puts the current job into the free list queue after calling the registered callback function.

Third, if there's no free streams available, gpufw_job_get_free() blocks the current program execution and waits for the MASTER_SIG_FREE_JOB signal from the GPU master thread instead of returning false directly.

Fourth, the rlevel ranges from 1 to 4 normally. But the rlevel will be added with 5 after the encoding operation to indicate that the same rlevel encoding operation has been done before. Thus, the excess encoding matrix buffer copies can be detected and avoided.

3.4 Cauchy Reed-Solomon Coding Layer

The Cauchy Reed-Solomon Coding layer is an abstraction layer upon the CrystalGPU framework, and is responsible to do CRS encoding/decoding setup and trigger the underlying encoding/decoding routines.

As illustrated in fig. 2 the Cauchy Reed-Solomon coding layer consists of 5 functions calls.

First, the *crs_init()* needs to be called before any other functions, and the number of streams need to be specified,

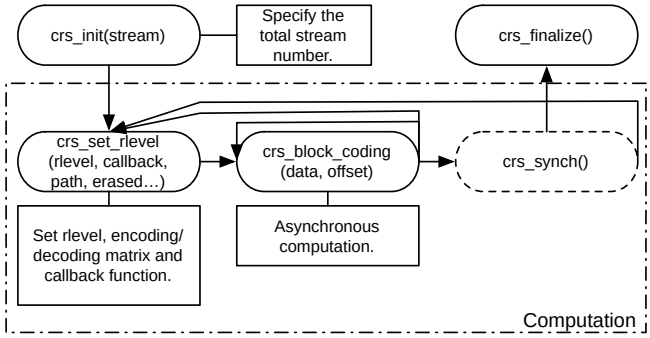


Fig. 2: Flow of CRS coding layer

or it will use the default stream number, 10. In this function, the main task is the memory allocation and GPU master thread initialization.

Second, the *crs_set_rlevel()* is called to do the reliability level configurations. If the current operation equals to CRS_DECODE, the argument erased, which is used to produce the decoding matrix, must be assigned. The erased[x] equals 1 if the x-th chunk is missing, and it equals 0 otherwise. The index of the data chunks ranges from 0 to 127, and the index of the checksum chunks ranges from 128 to 144, 160 or 192 which depends on r_level. Notice that the total number of erased units must be less or equal to the maximum erasure-tolerate number specified in the reliability level, or it will just return false to indicate failure.

Third, the *crs_block_coding()* is called to do the asynchronous computation by copying 512-KB input data to the input buffer of the CrystalGPU framework. If there's a free stream, this function call is non-blocking. Otherwise, it blocks the program execution as described in section 3.3. If the input data is larger than 512 KB, the *crs_block_coding()* can be called multiple times continuously to handle the encoding operation of different chunks in the same file.

To avoid excess data copies, the copies of the encoding matrix buffer will be skipped if the stream returned by gpufw_job_get_free() has the same r_level as the current crs_block_coding() operation.

Fourth, the *job_synch()* is used to block and wait for the last submitted job to be completed. Last, when the file server process (*afsfuse server*) is terminated, the *job_finalize()* is called to clean up the allocated memory and data structures.

3.5 AFS FUSE Layer

AFS FUSE layer is based on FUSE and Rx to implement the client-server model of CRSFS according to [13].

3.5.1 The Overview

As shown in fig. 3, a client process file request is first captured by the FUSE kernel module, *FUSEFS*, and forwarded to the /dev/fuse. Then, the user space daemon of FUSE, *afsfuse client*, reads command from /dev/fuse

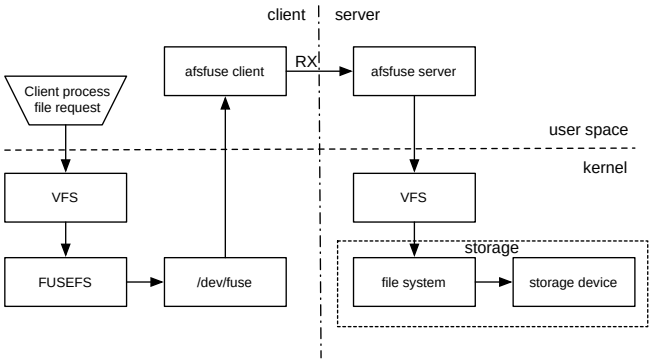


Fig. 3: Flow of AFS FUSE layer

Table 2: Hardware specifications

CPU	Intel® Xeon® CPU E5620 @ 2.4GHz 4C/8T × 2
RAM	4GB DDR3-1066MHz × 3
GPU	Nvdia® Tesla M2050 × 1
HDD	Hitachi 1.5TB 7200RPM × 2 (RAID 1)

Table 3: Software settings

OS	Ubuntu Server 10.04.3-amd64
Kernel	Linux 2.6.32-33
File System	client: ext4, server: ext4
CPU lib	Jerasure 1.2
CUDA	4.0

and calls the corresponding xmp_xxx functions according to the fuse_operations list registered by *afsfuse client*. The xmp_xxx functions then invoke the corresponding rxc_rxc_xxx functions in *afsfuse server* via Rx to actually access the storage device and do the accompanying CRS computations if the file written to has the extended file attribute named "*user.crsfs.rlevel*" and its value is valid as listed in table 1.

The only job of the client node—with *afsfuse client* running—is to forward these file requests captured by *FUSEFS* to the server node—with *afsfuse server* running—via Rx protocol. All the CRS encoding/decoding operations are performed on the server node. There could be a storage node if the storage device is outside of the server node, such as using an external iSCSI storage node.

3.5.2 Implementations in AFSFUSE Server

The important functions that trigger the CRS coding routines in *afsfuse server* are the followings: setxattr(), write() and read().

a) Setxattr: In setxattr(), whenever there's a value change at "*user.crsfs.rlevel*" and the new value is valid, *afsfuse server* will read all the contents of the file 512-KB chunk by 512-KB chunk and skip the last one if the file size is not a multiple of 512 KB.

b) Write: As for write(), due to the limitation of the linux kernel and FUSE, the maximum write size is 4 KB. Even if this limitation dose not exist, it can't be assured that the write file request size is always 512 KB. Therefore, the write request is buffered up in a 512-KB buffer on 512-KB chunk basis Whenever 512-KB chunk is buffered up, the CRS encoding operation will be triggered asynchronously.

c) Read: As for the read(), after *afsfuse server* performing the actual file accesses, if there's an I/O error, the CRS recovering procedure is triggered. Assume that the file can tolerate up to m-page erasures according to its rlevel.

First, *afsfuse server* reads the 512-KB data chunk and $4m$-KB checksum chunk page by page. If the page is read without the *EIO* error, *afsfuse server* copys it to the 512-KB buffer. Otherwise, *afsfuse server* marks the value of the int erased[x] to 1 where x is the index number of the current page in the big chunk as described in section 3.4. All the recovering routines need are the first 512-KB survived pages and the erased[$128 + m$] with the erased page indexes set to 1. Then, *afsfuse server* calls crs_set_rlevel() to create the decoding matrix and then invokes crs_coding() to submit the job to GPU as mentioned in section 3.4. After that, crs_synch() must be called before returning the values back to xmp_read() in *afsfuse client*.

After all the missing pages in the big chunk are recovered, *afsfuse server* just re-writes these pages at the same file offset. The underlying storage devices, such as hard disks, will re-map these logic block addresses (LBAs) of the bad sectors (or pages) to the new ones in the reserved area if they encounter I/O errors with write operations.

4. Evaluation

In this section, both of the GPU primitive for CRS coding and the file operations—setxattr, write, and read—on AFS FUSE layer are evaluated.

4.1 Experiment Environment

As listed in table 2, the computer used in this evaluation is a powerful single node computer which runs all the client, server and storage nodes.

The software settings are described in table 3. Notice that, the CPU coding library used as the baseline for comparison is the Jerasure version 1.2 released by James S. Plank [14]. The Jerasure routines for CRS coding are wrapped up as the same interface as the CRS Coding layer described in section 3.4 and are inserted at the same position in the AFS FUSE layer. Therefore, the Jerasure coding routines also benefit from avoiding excess encoding operations generation if rlevel of the current job is the same as the one of the last job.

Due to the limitation of both the linux kernel and the FUSE framework, each write operation can only write up to 4-KB data per write request. Any write system call with data size larger than 4 KB is split into 4-KB write requests automatically. Similarly, the read operation is limited to 128-KB data transfer per read request.

In order to test the performance of transparent recovery with maximum erasures, the errno is set to EIO selectively and the m erasures are distributed evenly into the 128 data units. This happens only when *afsfuse client* reads the first page of each 512-KB chunk, resulting in one recovery operation for each 512-KB chunk. The read evaluation is performed by directly reading from files with preset reliability levels and excludes the time writing the recovered pages back to the files.

All the evaluated results illustrated in the following graphs are the average values of 10 identical experiments. Each experiment is tested with all reliability levels (from 1 to 4) and file sizes (from 512 KB to 16 MB or 1 GB). All the GPU parts are tested with stream_count set to 10.

4.2 GPU Primitive for CRS Coding

Fig. 4 shows the performance of GPU Primitive for CRS coding. The *htod* stands for "host to device memory copy" and the *dtoh* stands for "device to host memory copy" similarly. The m, n in each test set mean the number of checksum rows (m) and the packet size (n). There are total $\frac{m}{\omega}$ checksum units and $\frac{k}{\omega}$ data units in one big chunk, where k is 1024 and ω is 8 in this work. Moreover, both tests of 4096 bits and 8192 bits packet size are performed to examine the possibility of increasing the erasure unit size from 4 KB to 8 KB. The memory copy size of the htod and dtoh are $k \times n$ bits (data transfer) and $m \times n$ bits (checksum transfer) respectively.

For the computation part, the performance of the (64, 1024) on 8192 bits packet size is much better than the one on 4096 bits packet size, which means the workload of the latter can't fully occupy GPU. And the throughput becomes nearly half from (m, k) to $(2m, k)$ as the complexity doubles from m to $2m$. Besides these tests, (16, 64) like RAID-6, including 2 checksum units and 8 data units, is also tested and reaches up to 14GB/s with large packet size. This shows that the GPU Primitive is quite efficient.

4.3 AFS FUSE Layer

The FUSE and the AFS FUSE introduce about 3.2 ms and 24 ms overhead per 512-KB chunk write system call respectively. The former is measured by launching a write system call with 512-KB data from a user space program, and directly returned from xmp_write() in *afsfuse client*. Similarly, the latter is measured by using the same operations, but this time, it's returned from rxc_rxc_write() in *afsfuse server* after receiving the arguments of xmp_write() in *afsfuse client* via Rx.

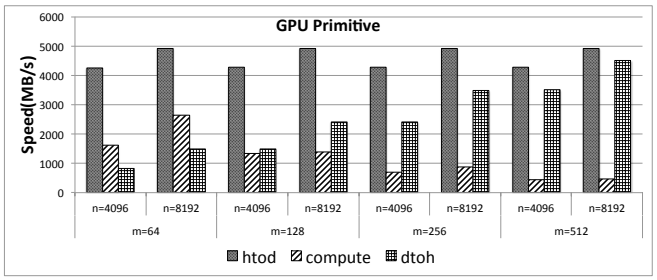
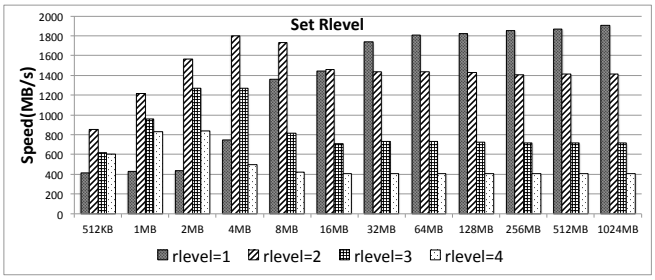
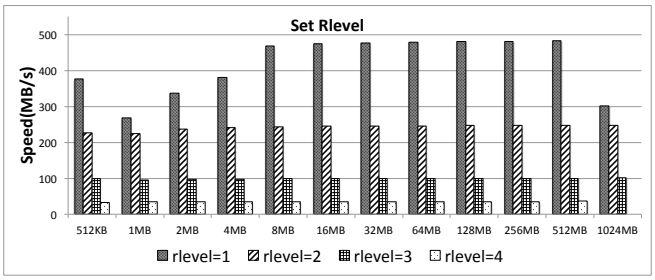


Fig. 4: Throughput of GPU primitive for CRS coding



(a)



(b)

Fig. 5: Throughput of setting different rlevels on: (a) GPU. (b) CPU.

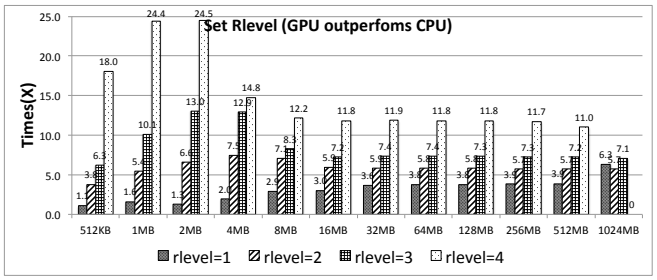


Fig. 6: Speed comparison on setting different rlevels between GPU and CPU

4.3.1 Performance of Setting Reliability Levels

As shown in fig. 5 and 6, the performance of the GPU encoding outperforms the CPU's, starting from 1.1x up to 24.5x. The results of these tests are all beyond the speed of the hard disk because all the tests are performed 10 times such that almost all files are cached in RAM.

All the performance results, except for rlevel=1, show the trend of first increasing and then decreasing to some stable value as data size increases. Some of them even outperform their counterparts in GPU primitive illustrated in fig. 4. That is the result of asynchronized operation functionality provided by CrystalGPU. By setting stream_count to 10 in this work, there are at most 10 CRS encoding/decoding jobs on the fly, which means that when the data size is less than or equal to 5 MB, *afsfuse server* at rxc_rxc_write() just reads 512-KB data from the file, copies them to the input buffer of CrystalGPU, invokes the encoding operation and then continues to encode the next 512-KB chunk *without any blocking delay*.

However, for all tests with rlevel=1, there's little benefit because the encoding task is so light such that before *afsfuse server* submits the next encoding job, the encoding job for previous chunk has already been finished.

Notice that, there's no test result in rlevel=4, 1024-MB file test on CPU because the setxattr system call has a time limit, around 14 seconds, which causes the test finished with the following error: *attr_set: Operation not permitted*.

4.3.2 Performance of Write Operations

The write performance on different size of files is also evaluated. It is tested by writing different size of data to the corresponding zero-length files with preset reliability levels. The result is shown in fig. 7. The tests with rlevel=0 are performed with no reliability levels.

As illustrated in fig. 7, the throughput of the write operations on GPU is quite stable due to asynchronized operation functionality. On the other hand, the throughput of the write operations on CPU decreases as rlevel increases because the CRS encoding operations on CPU are executed on the same thread as *afsfuse server*. Therefore, the request service time includes the encoding operation time, which hurts the performance.

All of the tests are limited by the maximum throughput in this framework. The throughput is mostly under 20 MB/s because the write system calls with data size larger than 4 KB are split into multiple 4 KB file requests as mentioned in section 4.1. Together with the round-trip overhead in the AFS FUSE framework, the throughput of the write operations becomes quite limited compared to the one of the read operations which can reach up to 120 MB/s. However, if there are multiple client nodes accessing the same server node, the total throughput can be aggregated to a larger one, reducing the impact of the low throughput for a single client node.

4.3.3 Performance of Transparent Recovery on Read Operations

The recovery throughput is quite low as shown in fig. 8 because the recovery procedure is a synchronized operation.

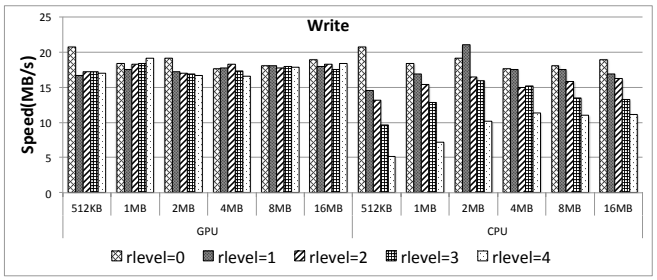


Fig. 7: Throughput of write operations.

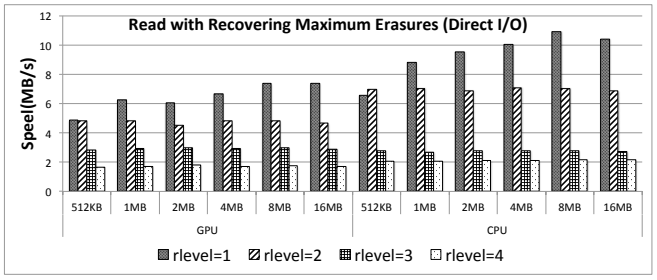


Fig. 8: Throughput of read operations with transparent recovery.

The client must block and wait until all the recovery operations finish, or this read operation fails as the data which the client requested for can't be served immediately. However, since I/O errors seldom happen, the request service time for the read operation is more important than the throughput of the recovery procedure. For a single 512-KB read operation with maximum erasures on GPU, the request service time for rlevel=1, 2, 3 and 4 are 47, 75, 126 and 233 ms respectively. When executed on CPU, these are 31, 39, 136 and 181 ms respectively.

Due to the memory transfer overhead and the synchronization of the recovery procedure, the performance of the GPU part is even worse than the one of the CPU part. However, for the tests of the rlevel=4, the performance of the GPU part is much the same as the one of the CPU part because the GPU with more computing power takes the advantage of less computation time compared to the CPU as the complexity increases.

5. Conclusion

In this work, the original file-wise reliability scheme is revised to cope with the larger page size in storage devices nowadays, and a prototype file system, *CRSFS*, is proposed to implement it with GPU acceleration and flexible design.

GPU primitive for CRS coding is a CUDA program optimized for the Fermi architecture of Nvidia GPUs. Combined with CrystalGPU, it shows up to 24.5x performance gains compared to the CPU counterpart in AFS FUSE layer. Therefore, it's clearly shown that the speed of CRS

encoding/decoding operations is no longer the performance bottleneck.

In addition to the GPU acceleration, CRSFS is also integrated with FUSE and Rx in AFS FUSE layer to provide high flexibility on storage system configurations. The client node in AFS FUSE layer provides POSIX file system APIs to client file requests via FUSE, which means that most programs can benefit from the flexible file-wise reliability scheme provided by CRSFS without rewriting their read/ write operations.

Acknowledgment

The authors would like to thank the support from NSC under grants 101-2219-E-007-006 and 100-2219-E-007-009.

References

[1] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1988, pp. 109–116.

[2] T.-T. Tseng and Y. Hsu, "A flexible and cost-effective file-wise reliability scheme for storage systems," in *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, ser. HPCC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 427–433.

[3] W. Krier and E. Liska, "FUSE design document," Sun Microsystems, Tech. Rep., 2009.

[4] "Rx: Extended remote procedure call." [Online]. Available: http://rmitz.org/rx/Rx.pdf

[5] A. Gharaibeh, S. Al-Kiswany, and M. Ripeanu, "Crystalgpu: Transparent and efficient utilization of gpu power," Networked Systems Lab, University of British Columbia, Tech. Rep. NetSysLab-TR-2010-01, 2010.

[6] M. Curry, A. Skjellum, H. Ward, and R. Brightwell, "Arbitrary dimension Reed-Solomon coding and decoding for extended RAID on gpus," in *Petascale Data Storage Workshop, 2008. PDSW '08. 3rd*, nov. 2008, pp. 1 –3.

[7] M. Curry, H. Ward, A. Skjellum, and R. Brightwell, "A lightweight, gpu-based software RAID system," in *Parallel Processing (ICPP), 2010 39th International Conference on*, sept. 2010, pp. 565 –572.

[8] A. Brinkmann and D. Eschweiler, "A microdriver architecture for error correcting codes inside the linux kernel," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 35:1–35:10.

[9] H. P. Anvin, "The mathematics of RAID-6," Tech. Rep., October 2007.

[10] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu, "A gpu accelerated storage system," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 167–178.

[11] T. Steinke, K. Peter, and S. Borchert, "Efficiency considerations of Cauchy Reed-Solomon implementations on accelerator and multi-core platforms," in *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*, July 2010.

[12] R. Nath, S. Tomov, and J. Dongarra, "An improved Magma gemm for Fermi graphics processing units," *Int. J. High Perform. Comput. Appl.*, vol. 24, pp. 511–515, November 2010.

[13] S. Singh, "Develop your own filesystem with FUSE," February 2006.

[14] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2," University of Tennessee, Tech. Rep. CS-08-627, August 2008.

On Survivability of Grouping Fault Detection in Large-scale Distributed System

Shuyu Chen[1], Huawei Lu[2], and Guiping Wang[2], and Xiaoyi Yuan[3]
[1]School of Software Engineering, Chongqing University, Chongqing, CHINA
[2]College of Computer Science, Chongqing University, Chongqing, CHINA
[3]Zhengda Software Polytechnic of Chongqing, Chongqing, CHINA

Abstract - *To implement fault detection under large-scale, high churn, strong reliability required network environments, a small proportion of members are likely to be more critical than others. The communications would be reduced significantly by removing them. We further the studies on GFDP (Grouping Fault Detection Protocol) and propose a Partitioning-avoidance GFDP (Pa-GFDP) to enhance the survivability of a detection system. Pa-GFDP could detect the critical members in the distributed system and neutralize them. Without global information, the accurate detection of critical members could be accomplished with little traffic overhead and within limited time threshold. Pa-GFDP was proven to be correct and effective by experiments.*

Keywords: Distributed System; Fault Detection; Survivability; Critical Member; Neutralization

1 Introduction

In recent years, the progress in deploying large-scale distributed systems such as Grid, Cloud, Wireless Sensor Networks (WSN) and P2P overlay networks, has been pretty fast. Related studies mainly focused on the subjects of efficient information transmission, integrity of functions, scalability and extensibility. But researches on the dependability of a system have drawn more and more attentions nowadays. David Patterson [1] has pointed out that the construction of today's computer system is to provide high-reliable network services. But the large open distributed systems suffer from threats of intrusions, attacks, component failures and so on, which make the systems undependable. How to deliver scalable and trustworthy network computing services using untrusted intermediaries becomes one of the most key tasks to build dependable distributed systems.

One of the most efficient ways to enhance the system's dependability is fault monitoring, while the fault detection is the fundamental component of fault monitoring. The traditional solutions of fault detection do not meet the requirements of modern large-scale distributed systems. The latest research achievements designed distributed approach to execute fault detection. As described in [7] and [8], the detectors formed a distributed system over the networks themselves.

For a large-scale distributed system, it is often not cost effective to protect all the nodes. Also, detectors in a distributed system are usually not equally important from the dependability point of view. Many researchers noticed the existence of critical links and critical nodes in the context of P2P overlay networks and WSNs. And in a distributed detection system, the same researches are necessary to enhance the survivability of the detection function.

An enhanced Grouping Fault Detection Protocol (GFDP [8]) called Partitioning avoidance GFDP (Pa-GFDP) is designed in this paper. And the highlights of Pa-GFDP are summarized as follows: 1) Local: Without global knowledge, a detector can identify whether it is a critical one based solely on local information; 2) Adaptive accuracy: Using limited information to estimate a critical member with high probability; 3) Traffic lightweight: No need to insert any extra traffic into the network; 4) Dynamic: As detectors could come and leave frequently, Pa-GFDP is able to run at each detector at any time to identify itself if it becomes a critical one at the given time and neutralize it.

The remainder of this paper is organized as follow. Section 2 discusses related works, the system model and definitions are introduced in section3, the protocol is discussed in section4. Experiments and conclusions are presented in section 5 and 6 respectively.

2 Related work

Highly reliable application systems which support fault detection based on fast developing Grid, P2P, and Wireless Sensor Networks systems have been brought out [9-11]. But traditional fault detection methods could not meet the requirements for modern networks. So several fault detection algorithms were brought out to satisfy these requirements such as large-scale, strong-dynamics and transmission-uncertainty [12-14]. And most of these algorithms are based on static heart-beat detection, but which could not quite meet the requirement of dynamics. A dynamic heart-beat fault detection based on grey model which efficiently reduces the

observed sample size is presented in [9], it solved the problem of dynamics in distributed systems, but it needs a large sample size, and with the problem of large network overhead.

Renesse [15] proposed a Gossip-style fault detection protocol to solve the problem of probably network congestion during messages dissemination using the fault detection based on time prediction. We furthered the theory and developed GFDP [8] and Bt-GFDP [7]. This protocol takes the advantage of the high reliability of message dissemination in the network while avoids the problem of network congestion, meanwhile the redundant messages were controlled in the system.

In our former works [7] and [8], when the detectors probabilistically leave and re-join the system, the detection structure becomes unstable. According to the observation of Saroiu et al. [3], to stabilize the detection system, cut vertexes and links should be neutralized.

3 Definitions and Limitations

A network monitoring system consists of a set of multiple members with limited size:

$$\Pi = \{m_1, m_2, \cdots, m_i\}, \text{ for } i > 2 \text{ and } i \in N$$

m_i in Π here is an abstract presentation of a module or a process, or even a node in the monitoring system. And members may be alive or failed, in other words, the member in the system may join in or leave out randomly.

The set of fault detectors in the system is defined as:

$$\Omega = \{d_1, d_2, \cdots, d_j\}, \text{ for } j > 2 \text{ and } j \in N$$

Any m_i in Π has a corresponding d_j exists in Ω, d_j is called a fault detector attached to m_i. An active member means its detector participates in the monitoring system; while a failed one means its detector may be crashed or detached. In the following description, we use "member" and its attached "detector" as the same meaning.

The following are the definitions used in this specification.

Component: A component is a maximally connected subgraph.

CM: Short for Critical Member, also refer as Cut Vertex. A CM is a vertex of a graph such that removal of it causes an increase in the number of connected components.

Bridge: A bridge, or cut edge, is an edge whose removal disconnects a graph.

K-connected: If it is always possible to establish a path from any vertex to all others even after removing any (k-1) vertices, then the graph is said to be k-connected.

Block: also called bi-connected component, a block is a maximally connected subgraph having no CM.

Three possible cases are illustrated in Fig.1, and CMs are colored in grey. In Fig.1(a), the CM is the joint of a 2-connected subgragh and a bridge; in (b), the CM is the joint of two 2-connected subgraphs; and in (c), the CM in the middle is the joint of two bridges. Indeed, all CMs fall into the three cases.

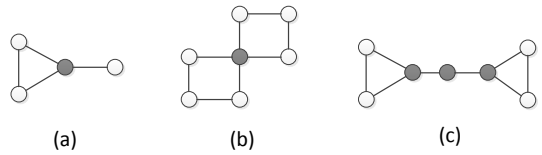


Fig.1 Three cases of CM

In this paper, we would consider the case of cut vertex. Bridges will not be considered. As removal either a vertex of a bridge would disconnect a graph.

4 Protocol specification

This section describes our distributed approach to identify the CMs under a large-scale distributed high-churn environment. The communication between members is derived from GFDP [8], mainly by gossiping. With the help of information inter-changed between members, we then design a zero-overhead passive detection method to identify CMs. The accuracy of the detection can be reinforced by an active detection method with a fairly low cost. Both passive and active detections are new functions added into Pa-GFDP, and both detections will be executed periodically to reflect the topologically changed of detection system.

4.1 Passive CM Detection

In the detection system which running Pa-GFDP, a message is propagated from one detector to others hop-by-hop using the gossiping mechanism. The forwarding message contains the path information from the starting detector to the current one. We propose to utilize such information to find CMs by sorting the neighbors of a detector into one or more blocks. According to the aforementioned definition, a detector is denoted as a CM if its neighbors belong to multiple blocks. In our scheme, we assume the following characteristics of gossiping in the monitoring system.

(1) A detector forwards the received message to part of its local view neighbors except the one where the message came from. And the number of forwarding destinations is determined by the configured gossiping fan-out, we denoted as c here.

(2) Each message is assigned a globally unique message ID. As every detector in the monitoring system only maintains a local view of its own, it does not have global

knowledge of the system. And as every detector in the system has a unique ID $d_i.id$, and it has an aforementioned *Beat Counter*, a message could be generated by combining $d_i.id$ with *Beat Counter*. Here we denote it as *Msg.id*.

(3) A detector gossips each message only once. If it receives the same message later, it simply drops the duplicates.

In the passive detection, each member keeps track of recently received messages. A list of records is cached on each member, called *MsgList*, with each entry representing a message received in the format of < *Msg.id*, list of $d_i.id$ that the message has traversed members>. A member also flags its neighbors their block information. At the beginning of a passive detection period, *MsgList* is empty and all neighbors of a member are assumed to be in different blocks. During the periodical execution of passive detection, a member randomly receives messages from its neighbors. By examining the path information of the received messages, it could discover circles formed by its neighbors. Thus a member could deduce that all neighbors on a circle belong to a block including itself. When two circles shares one or more members exclusive of itself, then it could deduce that the members on both circles are in the same block. This is how blocks merging and flags changing in a member's local view. As more messages arrive, member keeps merging the blocks and changing the block flags of its neighbors. The pseudo code of *On_Recieving*() is illustrated in Fig.2.

```
/* msg: a gossiping message;
msg.TTL: a pre-configured hops times that msg can be
     forwarded;
msg.id: the message ID of msg;
N: the detector receiving msg.*/
1.  On_Receiving( )
2.  {
3.   if( N receives msg for the 1st time )
4.     N creates a new entry in N.MsgList;
5.     if( msg.TTL > 0 )
6.       Gossip(msg); //forwarding the msg
7.   }
8.   else  //N has received the msg before
9.   {
10.    N creates a new entry in N.MsgList;
11.    FindEntry( N.MsgList, Msg.id ); //Find the entries
                                       //with same Msg.id
12.    MergeBlk( N.LocalView );  //Merge blocks and
                                 //flag Local view
13.    N drops msg;
14.  }
15. }
```

Fig.2 Pseudo code of function *On_Receiving*()

After running a determined period of passive CM detection, all the neighbors of a member will probably be merged into a few blocks. If there are only one block flag in a member's local view, it is not a CM. Otherwise; an active detection process is triggered for further determination, which is introduced in 4.2. Note that the passive detection is executed passively during the process of message dissemination. All the information is attached in the normal messages, so it would not incur any additional traffic overhead.

4.2 Active CM Detection

With the passive detection, a member knows for sure that it is not a CM if all the neighbors belong to a single block. However, having two or more blocks remaining at the end of passive detection does not mean a member is a CM. For example, a node might not be able to receive messages containing all possible paths from its neighbors because the messages are forwarded in a manner of gossip, or the message *TTL* threshold is reached. In order to identify CMs, an active detection is necessary. Compared to the passive detection, the active detection achieves shorter convergence time at the cost of additional but acceptable traffic overhead.

If a member's block flags are not consistent after a long-enough period of passive detection, which means that the neighbors of that member are sorted into two or more blocks, it regards itself as a suspect of CM and immediately starts an active detection process.

At first, it randomly selects a neighbor from each block and numbers the each neighbor with a unique *Block-index* (e.g. 1, 2, 3...). Then the node sends probe messages to these neighbors. The format of the probe message is < $d_i.id$, *Msg.id*, *TTL*, *Block-index*>, where $d_i.id$ is the suspect's member ID, *Msg.id* keeps the records the time the probe message is generated, *TTL* (time to live) is a pre-configured number of hops that the message can be forwarded, and *Block-index* denotes the index of the block to which the suspect sends the probe message. Each member keeps a probe message list. There is one entry for each suspect in the connection list with the format of <suspect's $d_i.id$, *Msg.id*, *Block-index* 1, *Block-index* 2...>.

Upon receiving a probe message, one of the following situations may arise.

(1) The member has already received the message, or the *Msg.id* of the message is smaller than that stored in the corresponding probe message list entry. The member just drops the message.

(2) There is no entry for the suspect that issues this probe message. The member creates a new entry for it.

(3) The *Msg.id* in the received message is larger than that stored in the corresponding probe message list entry. The suspect replaces the older *Msg.id* and *Block-indexes* stored in the probe message list with the new one.

(4) The *Msg.id* of the received message is the same as the one stored in the corresponding probe message list entry but the *Block-index* of the message is not the same. The node

adds the new *Block-index* to the corresponding entry and sends an arrival message back to the suspect. The arrival message therefore contains d_i.*id* of the current member, two or more *Block-indexes*, and the *Msg.id* stored in the entry. A member does not send any arrival messages until it receives at least two probe messages with different *Block-indexes*.

In this way, the neighbors of a CM suspect can be merged into fewer and fewer blocks. If only one block remains in its local view, the suspect is not a CM. Otherwise; the suspect must be a CM. In the case of being verified a CM. a progress of CM neutralization would be trigger, which is discussed below. Since the initial *TTL* value of a probe message is usually small, we are able to get the result of the active detection much sooner than the passive detection. In other words, the active detection can be applied as a useful complement to the passive detection. It can also be utilized as an independent approach to identify CMs if we value speed over cost.

4.3 CM Neutralization

The goal of CM neutralization is to enhance the system survivability with respect to topology connectivity. CM neutralization is relatively easy to achieve by building extra connections between members in different blocks. After the new connection is built, two initially independent blocks are merged into one block. Consequently, all members in the graph will get 2-connected and the CM becomes a normal node.

4.4 Traffic Overhead

We evaluate the traffic overhead by counting the messages delivered due to the CM detections. Note that gossiping is adopted in our basic GFDP as the basic mechanism for data dissemination. Even if there is no passive detection, the traffic overhead of gossiping does exist as an element of system running. Hence the passive detection does not incur any additional traffic overhead because it only utilizes the information extracted from the existing messages. For the active detection, suppose the system has n members, let ***c*** be the gossiping fan-out and let ***t*** be the initial *TTL* value.

Note that a detector will not forward the probe message if it has already sent an arrival message back to the corresponding suspect. We define the *traversal set* as the members which are traversed by the same block number of a suspect's probe message. And the traversal sets of different blocks of a suspect will not overlap. As a result, the total traffic overhead of probe messages is $O(n^2c/2)$, where $nc/2$ is the number of edges in the whole graph. On the other hand, the traffic overhead is also limited by the initial *TTL* value. It can never exceed $O(nc^t)$. Therefore the total traffic overhead of forwarding probe messages is $min(O(n^2c/2), O(nc^t))$. In the detection system, the value of c is much smaller than that of n. The inequality $c^t \leqslant n$ holds when the initial *TTL* value t is limited to save traffic cost. Thus we can conclude that the total traffic overhead of active detection is $O(nc^t)$.

5 Experiments

We make use of the test environment of [8], but we changed some configurations to modify the topology of the detection system. Altogether we have generated 10 different initial configurations with system size from 500 to 5000. 3of them are selected as representative for the subsequent experiments, their initial configurations are listed in Tab.1.

Tab.1 Information of different configurations

Round	System Size (n)	Gossip Fan-out (c)	Number of CMs	TTL (t)
1	1000	2	100	5
2	1500	4	150	5
3	3000	4	300	10

In the series of experiments we first evaluate the accuracy of CMs detection. The result is shown in Fig.3. From the result we can see that the gossiping fan-out would effect on the efficiency of CMs detection. But larger t would incur much higher traffic overhead. Moreover, larger *TTL* has less effect at last several rounds.

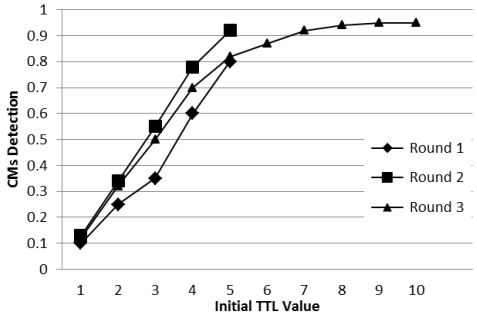


Fig.3 Accuracy of CMs Detection

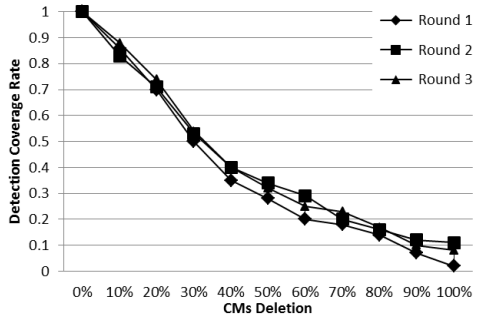
We evaluate the survivability of the detection system by using GFDP and Pa-GFDP comparatively. The result is shown in Fig.4. In both experiments we deleted the CMs at a fixed rate while GFDP and Pa-GFDP were running. From the test result we can see that Pa-GFDP improves the survivability of the system remarkably.

6 Conclusions

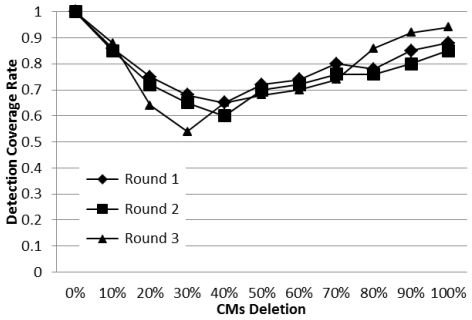
The connectivity among members basically determines the survivability of communications in large-scale distributed systems. It is observed that a small portion of members are more critical to the system than others. Removal of the critical ones would destroy the topology of a connected graph. To further our work on GFDP, we propose a partition avoidance approach to identify critical members and neutralize them. Compare to GFDP, the proposed Pa-GFDP mainly appends three functions: passive CM detection, active CM detection

and CM neutralization. From the experimental results, we can discover that by deploying Pa-GFDP we could enhance the survivability of the system significantly.

Moreover, by little modification, the methodologies proposed in this paper could be used in many large-scale distributed systems, like WSN, P2P overlay networks, to identify the cut vertexes and cure them.



(a) GFDP



(b) Pa-GFDP

Fig.4 CMs deletion Impact on Detection Coverage

7 References

[1] Patterson, D. "Recovery Oriented Computing" [M]. Retrieved from http://roc.cs.berkeley.edu 2006.

[2] Xun Wang, Sriram Chellappan, Phillip Boyer, Dong Xuan. On the effectiveness of secure overlay forwarding systems under intelligent distributed DoS attacks, IEEE Transactions on Parallel and Distributed Systems, 17(7), 619–632, 2006

[3] S. Saroiu, P. Gummadi, S. Gribble, A measurement study of peer-to-peer file sharing systems, in: Proceedings of Multimedia Computingband Networking (MMCN) San Jose, CA, USA, 2002.

[4] P. Keyani, B. Larson, M. Senthil, Peer pressure. Distributed recovery from attacks in peer-to-peer systems. in: Proceedings of IFIP Workshop on Peer-to-Peer Computing, Pisa, Italy, 2002.

[5] S. Servetto, G. Barenechea. Constrained random walks on random graphs: routing algorithms for large scale wireless sensor networks. in: Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications, Atlanta, GA, USA, 2002.

[6] G. Liu, C. Ji. Scalability of network-failure resilience: analysis using multi-layer probabilistic graphical models, IEEE/ACM Transactions on Networking, 17(1), 319–331, 2009.

[7] Huawei Lu, Shuyu Chen, Xiaoqin Zhang, Guanghui Chang. Byzantine-Tolerant Grouping Fault Detection Protocol under High Churn Networks, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2011), Las Vegas, Nevada, USA. 2011.

[8] Guanghui Chang, Huawei Lu, Shuyu Chen, Ishiang Shih. Grouping Fault Detection Protocol under Dynamic Network Environments. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2010, pp. 151-155, Las Vegas, Nevada, USA. 2010.

[9] M Yamanouchi, S Matsuura, H Sunahara. A fault detection system for large scale sensor networks considering reliability of sensor data, Proc of the Ninth Annual International Symposium on Applications and Internet (SAINT'09), 255-258, 2009.

[10] H M Lee, D S Park, M Hong, et al. A Resource Management System for Fault Tolerance in Grid Computing, Proc of International Conference on Computational Science and Engineering (CSE'09), 609-614, Feb. 2009.

[11] M Chtepen, F Claeys, B Dhoedt, et al. Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids, IEEE Transactions on Parallel and Distributed Systems., Vol.20, Issue No.2, 180-190, 2009

[12] P Stelling, I Foster, et al. A fault detection service for wide area distributed computations, Proc of The Seventh International Symposium on High Performance Distributed Computing, 268-278, Jul. 1998

[13] S. Saroiu, P. Gummadi, S. Gribble. A measurement study of peer-to-peer file sharing systems, in: Proceedings of Multimedia Computing and Networking (MMCN) San Jose, CA, USA, 2002.

[14] A Jain, R K Shyamasundar. Failure detection and membership in grid environments, Proc of the 5th IEEE/ACM Int'l Workshop on Grid Computing (GRID'04). Los Alamitos, CA: IEEE Computer Society Press, 44-52, 2004.

[15] R Renesse, Y Minsky, M Hayden. "A gossip-style failure detection service", Proc of International Conference of Distributed Systems Platforms and Open Distributed Processing (IFIP), 2000.

AN APPROACH FOR FAULT MANAGEMENT BASED ON AUTONOMIC COMPUTING PLUS MOBILE AGENTS

Sergio Armando Gutiérrez, Ms.Eng[1], John Willian Branch, PhD[2]
[12]Facultad de Minas, Universidad Nacional de Colombia, Sede Medellín
Email: saguti|jwbranch@unal.edu.co

Abstract - *In this paper, a new approach for Fault Management, based on the autonomic computing paradigm and mobile agents technology is introduced. The main features of this approach are discussed, its main benefits are presented and some result of its application are evaluated..*

Keywords: Autonomic computing, mobile agents, expert systems, fault management

1 Introduction

The capacity of detecting failures in a given system is a very important challenge which arises in many engineering disciplines. When dealing with expensive equipment's, large and complex systems, or an infrastructure on which critical services or processes depend, requirements of availability and reliability increase dramatically. This is a reason why fault management becomes a very important research topic in areas such as chemical, aerospatial, nuclear, and electrical engineering, among many others [1]. An emergent field in which expensive equipment and critical systems arise is telecommunications (TELCO). Today's TELCO networks are very complex systems involving hundreds and even thousands of components, every one having multiple variables and features to be evaluated in order to assess how well the services running on them are performing. Due to need of quality service, and because of the growing dependency that organizations and people have on these services, networks (and the infrastructure and platforms where services run), need to have a high degree of availability as well as the capacity to react to failures and problems which might arise during their operation. Every day, critical mission is becoming a more commonly used term for designating the infrastructure which is used to run TELCO services. Critical mission means that systems should be available 100% of the time, and must not be interrupted under any circumstance, as an interruption might mean that a critical process cannot be performed [2]. This requirement of high availability implies that any event associated to failures in any component of the system be managed in such a way that the degradation or unavailability of a system be avoided or minimized as much as possible. This is the reason why fault management has become a big issue in the field of operation and engineering. As in many cases, the ability of human operators and supervisors is not quick and efficient enough to respond in the best way to the event of a failure. This work presents an alternative to conventional fault management based on Expert Systems by using two emergent technologies as Mobile Agents and Autonomic Computing

2 Fundamentals

2.1 Autonomic Computing

Autonomic Computing is a concept inherited from bioinspired computing in its beginnings. The first work where Autonomy Oriented Computing is introduced is the work by Liu et al [3]. This work defines the fundamentals of Autonomy in computing and the four main characteristics which should be exhibited by a computing system according to this paradigm:

• **Autonomy:** The entities in the system are rational individuals, which are capable to act in an independent way; in other words, the system does not have a central component and manager.

• **Emergent:** When system entities cooperate and work together, they exhibit behaviors not available or possible to obtain by individual entities separately.

• **Adaptive:** System components are capable to modify their behavior according to changes present in the environment where system operates.

• **Self-organized:** The system components are capable to organize themselves to achieve the previous commented behaviors. This paper also presents the several types of Autonomy Oriented Computing according to how autonomy is achieved by the system.

In Jin and Liu [4], Autonomy Oriented Computing is formally defined, by employing set notation to express the concepts associated: Environment, computing entity, state, behavior, goal are some of the concepts which are expressed in a formal language. From a more applied point of view, and again from a bioinspired perspective, Horn [5] presents the approach to Autonomic Computing from perspective of a industry leader as IBM.

Following Horn, there are eight keys elements which has an Autonomic Computing System:

• **Self-Awareness:** This is, the system should know itself. It will need detailed knowledge of its components, current status, ultimate capacity, and all connections with other systems to govern itself. It will need to know the extent of its "owned" resources, those it can borrow or lend, and those that can be shared or should be isolated.

• **Self-Configuration:** System configuration or "setup" must occur automatically. Also, the system must modify itself, in such a way that its configuration be the most adequate to cope with environment conditions.

• **Self-Optimization:** This is, the system will always try to find other ways to improve its functioning. It will monitor its constituent parts and fine-tune workflow to achieve predetermined system goals.

• **Self-Healing:** System must be able to recover from events that might cause malfunctioning. The system also must be able to detect problem or potential problems, and according to this, define alternate ways to perform, applying reconfiguration to keep the system working.

• **Self-Protection**: Starting from the fact of the potentially aggressive and hostile environment where system resides, it must be able to detect, protect and identify potential attacks or vulnerabilities, so that it can protect itself and maintaining working in a secure and consistent state.

• **Self-adaptability:** This is, the system must know its environment, the context which surrounds it when operating, and the other entities cohabitating with it. It must be able to adapt to this environment, and its changing conditions, by reconfiguring itself or optimizing itself.

• **Openness:** The components in system must be open to communicate each other, and must be able to work with shared technologies. Proprietary solutions are not compatible with Autonomic Computing philosophy.

• **Self-containment:** Components within an Autonomic Computing System must be able to perform the task or tasks they have assigned, not requiring external interventions for the performing itself, and hiding the complexity to end user.

Complementing the work by Horn, Lin et al [6] review Autonomic Computing from the perspective of Software Engineering. They present a proposal of metrics which could be used to evaluate the quality of frameworks based on Autonomic Computing. In other side, the works of Sterritt [7], Magedanz et al [8], Ionescu et al [9] and Tizghadam et al [10] evidentiate how Autonomic Computing has become a very important research topic in the field of Information Technologies, both for academic communities and for industries, in solving many problems which are becoming difficult to face with traditional and conventional approaches.

2.2 Mobile Agents

Agents are enjoying a lot of popularity as a novel abstraction for structuring distributed applications. It is a technology from the field of Artificial Intelligence. Although, there is not a precise and widely adopted definition for Agents, the tendency is define it through the features they should expose [11].

Following to Yubao and Renyuan [12], an agent is an entity possessing the following characteristics:
• **Self-government:** Agents should have the ability to governate themselves, without external interference from the outside world while they are performing their tasks.
• **Smart:** Agents should implement certain functions and be able to choose the required information to complete their tasks. They also should be able to get knowledge from the performing of their tasks.
• **Lasting:** The agents should survive, according to their participation in the tasks.
• **Co-relation:** This is the social behavior defined by theoretical definitions. In real world, the cooperation is presented as messages exchange among the agents in the system.

Figure 1 presents a classification of software agents according their functions, their properties, or other relevant features which can be assigned to them. Specifically, a feature relevant in the approach to be proposed is mobility. That is, the capacity of the agent to move itself autonomously across the environment where it runs, with the goal to execute its designated task at other locations where be required.

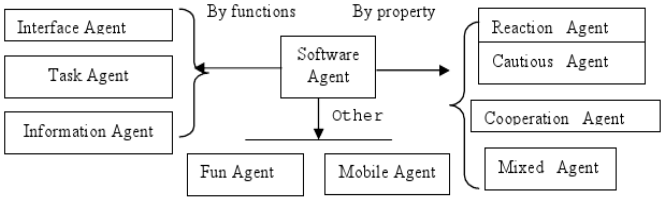


Figure 1: Software Agent Classification [12]

3 Proposed Architecture

For the goal of offering high availability and reliability on critical systems, increasing the reliability of Fault Management (FM) component or components is a very important goal to achieve. The elements detecting Faults, with the goal of detect them, isolating them or event correct them should become very smart and reliable, and should adapt to the complexity of the systems where they are meant to be used.

The approach to FM which is proposed consists in implementing Mobile Agents (MA), which exhibit the eight fundamental features of AC defined by Horn [Horn, 2001]. These agents will be designated to monitor specific resources, and will use a Knowledge Database which will store criteria defining faults for the associated resource. According to these criteria, the agents will be able to anticipate to faults on the component, by analyzing and detecting patterns on the designated assess metrics for the resource and by means of mobility feature, agents would be able to move across the system to monitor the designated resource at other locations.
The architecture for the proposed approach is composed by the following components:
• Knowledge Database (KDB): Is a database containing the criteria which are going to use by the agents to determine faults and how to react before them.
• Data Collector (DC): It is a component which will receive the data coming from the agents, and will send them information regarding changes in the criteria for fault detection.
• Mobile Agent (MA): It is an agent which will perform the actions on the resource and will traverse the network monitoring similar resources in other hosts.
• Managed Resource (MR): It is the resource in the hosts which will be supervised by the agents, for possible faults and problems in its performance. This resource can be a hardware element, a software element, or the behavior of a particular metric or behavior.

Figure 2 illustrates the architecture.

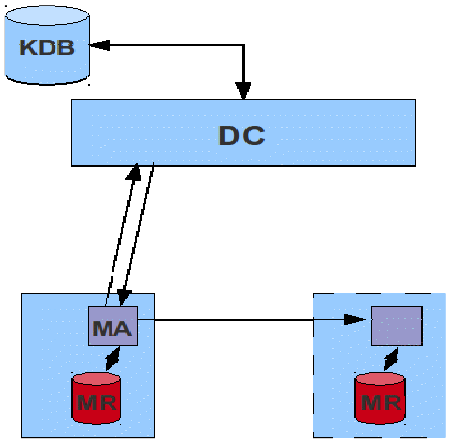


Figure 2: Proposed approach architecture.

Database as mentioned, will have two purposes. First, it will be to contain the data reported by the agents, for example, whenever it performs a correction because of a failure, or according to periodic monitoring if defined. The other purpose will be to store rules or instructions defining the criteria to act when a faulty behavior is detected. For example, if a failure is a filesystem full, the criteria could be what files to delete, or what kind of files to delete. Also, according to the functionality which performs the system component, the collected data would be useful to predict that a failure is yet to come, and so, indicate a proactive action triggered on the MA.
Data collector would be the entity which communicates with the MA. It will have two purposes. The first, collect the data from the MA, by receiving on them, or by triggering a data query to them. The other will be to perform smart analysis on data collected, and if possible, to modify the criteria of fault detection and notify these changes on criteria to MA. For example, if MR is a filesystem, the data collector could analyze the growing rate of a particular file and predict that it might cause a failure by full filesystem and notify this to MA. Also, the data collector would be useful to notify exceptions to MA, so that they could ignore the application of a criteria. The DC could be an agent itself. Mobile Agent would be the entity performing the detection and correction of failures on the hosts. It would be implemented in such a way to apply criteria to solve faults in a particular resource, and it could choose to travel or to clone itself to send a copy of itself to another host requiring supervision in the same resource. Managed Resource would be the system component under supervision and control by Mobile Agents. It might be a hardware or software element, or even a service or task.

4 Experiments and Results

The present section describes de tests performed to validate the advantages of the proposed approach for fault management. For these tests, a preliminary implementation of the proposed approach was performed, and run on a typical platform for the voice over IP Service. Two aspects were assessed. The time required to detect a failure, and the bandwidth consumed.
For the validation of the approach proposed in this work, a typical platform in the Voice over IP Services was chosen. This kind of platform was chosen, because of its criticity, and the relationships established between the components, which tend to induce failures in the other components, when a problem occurs.
The figure 3 depicts the testbed which was used for the implementation.

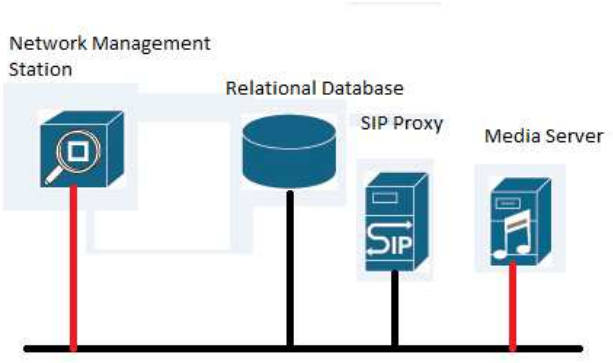


Figure 3: Testbed platform

For the implementation of the agents, the used tool was JADE, a very popular framework for implementing Mobile Agents on Java Programming Language.

For the testing, three scenarios were defined and tested. The three scenarios consist in inducing failures on a Resource node, and measuring the time used to detect (and correct) the failure.

4.1 Full filesystem event

On the first scenario, the first failure to be induced was a filesystem full on Relational Database. The failure was induced by creating an increasing size file, by means of an infinite loop script which concatenated 1MB files to an existing file until the whole filesystem was filled.

The SNMP based detection was performed by running a shell script which invocated snmpwalk system command once every second, querying the OID referencing the filesystem to be monitored. For the agent scenario, platform was started up, and a monitor agent was created on the supervised host, indicating it what filesystem to monitor. For this first test, no action was defined for the monitoring agent at startup time. At failure occurrence, DCA instructed the agent the action to perform, which in in this case was deleting the file causing the filesystem full.

Table 1 presents the results of the execution of this test scenario.

Variable	SNMP TEST	AGENT TEST
Elapsed time	3 mins, 42 sec	4 mins, 4 sec
Total packets	811	318
Average Bandwidth (Mbps/s)	0.003	0.002
Average packet size (Bytes)	87.893	155,082
Average packets per second	3,642	1,303

Table 1 : Data for the test of full filesystem fault at database

4.2 Dead system process

The second test scenario is the detection of a dead process within the SIP Proxy. The failure is induced by killing the process, making it disappears from system processes table.

The SNMP based detection was performed by running a shell script which invocated snmpwalk system command once every second, querying the OID referencing the entry of the process within the system process table exposed through SNMP. For the agent scenario, platform was started up, and a monitor agent was created on the supervised host, indicating the name of the executable associated to process. In this case, a single process was monitored, although the approach is valid for multiple instances of same process. For this test, no action was defined for the monitoring agent at start time. At failure occurrence, DCA instructed the monitoring agent to restart the process.

Table 2 presents the results of the test for a system dead process.

Variable	SNMP TEST	AGENT TEST
Elapsed time	1 min	1 min
Total packets	508	147
Average Bandwidth (Mbps/s)	0.005	0.003
Average packet size (Bytes)	87.893	155,082
Average packets per second	8,47	2.45

Table 2: Data of the test of dead system process at SIP Server

4.3 Network interface missconfiguration

The second test scenario is the detection of a missconfiguration within the Mediaserver. The failure is induced by modifying the network interface configuration, changing the duplex mode from full to half. This is a very common error on this kind of equipments, and usually causes malfunction in the service it offers.

The SNMP based detection was performed by running a shell script which invocated snmpwalk system command once every second, querying the OID referencing the duplex mode of the main network interface of the mediaserver. In this case, the monitoring script was capable, after detecting the error, to perform a correction by setting the parameter of duplex mode, by executing snmpset system command. For the agent scenario, platform was started up, and a monitor agent was created on the supervised host, indicating the name of the main network interface in the host.

For this test, no action was defined for the monitoring agent at start time. At failure occurrence, DCA instructed the monitoring agent to reconfigure the network interface.

Table 3 presents the results of the execution of this test scenario.

Variable	SNMP TEST	AGENT TEST
Elapsed time	2 min	2 min, 32 secs
Total packets	203	84
Average Bandwidth (Mbps/s)	0.001	0.0006
Average packet size (Bytes)	87.893	155,082
Average packets per second	1,69	0,55

Table 3: Data of the test of network interface missconfiguration at mediaserver.

5 Results Discussion

In this section, a discussion on the results obtained from the tests will be presented. A very important point to state is the fact that in the first two tests, besides of the fault detection, from the information that DCA returns at failure occurrence, the agent is capable to perform an action to correct the fault condition. The concept behind the implemented platform is providing the agents with logic and actions (behaviors) allowing that, from the information provided by the DCA in this case, they be capable to apply corrections.

In the standard implementation, SNMP based detection is not capable to correct complex faults as it should have external mechanisms to apply any correction. SNMP can solve very specific faults, if the correction consists in modifying a single variable, as seen on the third scenario, where the script could restore the network interface configuration. In the other cases, an autonomous correction y means of SNMP is not possible, and the intervention of a human operator is required. The fault correction is reactive, not proactive.

The agents implementation, by default is capable to provide mechanisms to apply corrections when particular types of faults are detected. Also, because of the local memory the agents have, they can try to apply corrections when faults are detected, without the need to ask for information to DCA, in an autonomous way. The agents, provided information to react to faults are capable to apply corrections in a proactive way. The traditional approaches, based on SNMP, are just based on notifications, and focused on alarm and notification triggering, leaving the action performed to correct the fault condition itself to an external entity (usually a human operator). It can be observed that the elapsed time, measured since monitoring was started until the fault condition was detected (and corrected), is a longer for agents than for SNMP. This can be explained as the agent platform (main container and satellital container) require some time for its initialization. In SNMP case, monitoring starts by just run the script executing snmpwalk. The average packet size is also greater for agents than SNMP because the network transport protocol that agent platform uses. JADE uses TCP for the communication, different to SNMP, which uses UDP. The usage of TCP as communication protocol in the agent platform increases the reliability of monitoring, as delivery of messages is guaranteed, and monitoring agents do not require to have additional logic for handling message delivery.

Another fact which can be observed is that agent approach requires less packets to be exchanged. As the agent communicates with DCA only for notifications and initialization tasks, and not for monitoring per se, less traffic is required. For SNMP, more packets are required, as the decision to detect the fault is taken at monitoring station, not in the monitored resource itself.

Briefing, from the point of view of the two variables assessed, the proposed approach, based on agents, shows to be more efficient, by exhibiting lower times to detect the fault, less traffic requirements, and besides, provides added value by the capability of executing and performing proactive corrective actions, to eliminate and avoid the fault conditions.

6 Conclusions

In this work, an approach for fault management using Autonomic Computing and Mobile Agents has been proposed. This approach ha showed to offer more advantages than conventional approaches by exhibiting the following features.

• From the architectural point of view, it is not server centric, as traditional systems, usually based on a master component which queries satelital agents to obtain information. Although the proposed approach has a master component, the monitoring agents do not require the communication with this component to perform their tasks, as they are designed to operate in autonomous way.

• It provides self-learning capacities. Traditional systems just report the fault by triggering an alarm when a threshold is reached, or when the fault itself arises, but they lack of efficient mechanisms to correct the fault. The agents in the proposed approach have the capacity to analyze causes of the detected problems, and perform further actions to avoid the fault occurrence.

• It provides a richer communication protocol among the involved entities, allowing having a detailed information regarding facts and actions to be performed. The concept of agents ontology provides a very robust mechanism for the communication protocol regarding message formats and interpretation. Traditional approaches are based on a simple request/response protocol, with very strict and limited message formats.

• It provides mobility, which allows to monitoring agents to move across the network to detect and correct faults. Traditional systems are based on static monitoring systems, which are confined to run at a single host. These features present a system which is very suitable for platforms as the ones used in Telecommunications service, which nowadays require high levels of availability because of the criticity of

the services they offer. Also, the implementation of the features previously mentioned contribute to improve the task of fault management, by providing added value to monitoring tasks, and correction for some of the limitations of traditional approaches.

7 References

[1] I. Hwang, S. Kim, Y. Kim, and C. E. Seah, "A survey of fault detection, isolation, and reconfiguration methods," IEEE Transactions on Control Systems Technology, vol. 18, pp. 636–653, 2010

[2] T. Fujisaki, M. Hamada, and K. Kageyama, "A scalable fault-tolerant network management system built using distributed object technology," in Proc. First International Enterprise Distributed Object Computing Workshop., 1997.

[3] J. Liu, K. Tsui, and J. Wu, "Introducing autonomy oriented computation," in Proceedings of First International Workshop on Autonomy Oriented Computation, 2001.

[4] X. L. Jin and J. M. Liu, Agents and Computational Autonomy: Potential, Risks, and Solutions., ch. From Individual Based Modeling to Autonomy Oriented Computation, pp. 151–169. University of Bath, 2004.

[5] P. Horn, "Autonomic computing: Ibm's perspective on the state of information technology," tech. rep., IBM Corp, 2001.

[6] P. Lin, A. MacArthur, and J. Leaney, "Defining autonomic computing: A software engineering perspective," in Proc. of the 2005 Australian Software Engineering Conference (ASWEC 05), 2005.

[7] Sterritt, R. (2001). Discovering rules for fault management. In Proc.Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2001. ECBS 2001.

[8] T. Magedanz, "Report on FET consultation meeting on communication paradigms for 2020," tech. Rep., Fraunhoffer FOKUS, 2004.

[9] D. Ionescu, B. Solomon, M. Litoiu, and M. Mihaescu, "A robust autonomic computing architecture for server virtualization," in International Conference on Intelligent Engineering Systems, 2008. INES 2008.

[10] A. Tizghadam and I. A. Leon-Garcia, "Autonomic traffic engineering for network robustness," IEEE journal on selected areas in communications, vol. 28, pp. 39–50, 2010.

[11] S. Franklin and A. Graesser, "It's an agent or just a program?: A taxonomy for autonomous agents," in Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages,, 1996.

[12] M. Yubao and D. Renyuan, "Mobile agent technology and its application in distributed data mining," in First International Workshop on Database Technology and Applications, 2009.

SESSION

SIMULATION AND MODELING + NUMERICAL METHODS

Chair(s)

TBA

Efficient data collection from Open Modeling Interface (OpenMI) components

Tom Bulatewicz, Daniel Andresen
{tombz@ksu.edu, dan@ksu.edu}
Dept. of Computing and Information Sciences, Kansas State University, Manhattan, KS, USA

Abstract— *The management of output data from simulation models can be simplified in grid environments by automating and standardizing the way in which they are collected and stored. In the context of component-based computer models with well-defined input-output interfaces, general-purpose data collector components can be linked to model components to retrieve output data and deliver them to online repositories via web services. We have developed a distributed data collector component that adheres to the Open Modeling Interface (OpenMI). The component buffers data to minimize the impact on a simulation's runtime and shares the buffer across compute nodes for load-balancing and cooperative delivery of data to web services. The buffering capability resulted in minimal runtimes within a single simulation and reduced data delivery latencies for concurrently executing simulations across a cluster. In this paper we report on the design and performance of the component.*

Keywords: OpenMI, data, web services, modeling, simulation

1. Introduction

The output data produced by environmental computer simulations often provides a starting point for investigation into the phenomena being studied. The data may need to be archived, aggregated, processed, and analyzed statistically or geographically before they can be visualized and interpreted. These may be performed by an individual or as part of a collaborative effort between groups within or across institutions.

Model output traditionally takes the form of local data files that may be of a custom format or adhere to simple standards such has comma separated values and extensible markup language or more complex standards such as netCDF and various database formats. Managing output files can be challenging, particularly in a grid environment in which simulations execute on multiple machines across a compute cluster.

As an alternative to data files, model output data can be immediately relayed to an Internet-connected data storage service that serves as a single repository for the data. This facilitates sharing of the data over the Internet and automates common tasks such as data archiving. In the general case, this capability is added to a model program by either modifying the model source code or incorporating intermediary software (often via scripting). In the case of model components with well-defined input-output interfaces, a general-purpose data collector component can be attached to any model component to retrieve the output data. Such components can mitigate data management challenges in any linked modeling context.

The Open Modeling Interface (OpenMI) [1] defines a standard way for software components to exchange data with each other and coordinate their execution. It defines a set of capabilities that a component must possess in order for it to be linkable to other components. These capabilities are both descriptive, to support the task of specifying component interactions at the domain level, and functional, to support the execution of a set of linked components. To fulfill the descriptive requirements, a component must be capable of providing a list (via a function call) of the domain quantities that it can provide and those that it uses as input, along with the units and spatial distribution of each. These are called *output exchange items* and *input exchange items*, and in the case of model components there is typically one output item for each quantity that it simulates and one input item for each of its inputs. To fulfill the functional requirements, a component must possess a *GetValues* function through which it provides data (that correspond to the exchange items) at runtime.

The GetValues function has three parameters that collectively identify a quantity at a single point in time at one or more locations as illustrated in Figure 1. A quantity (labeled Q in the figure) is represented by an object with several properties such as a textual identifier and units information. A time T is represented by a simple date object. A list of locations E is represented by an *elementset* object that contains a collection of *element* objects that each have a textual identifier, spatial shape (point, line, or polygon) and geographic coordinates. The GetValues function returns an array of real numbers called a *valueset* V such that each number corresponds to an element (based on its index) and collectively represents the state of a quantity at a point in time.

The GetValues function not only provides a means for the exchange of data between a group of linked components (called a *composition*) but it also provides a means for their coordinated execution at runtime. A special component

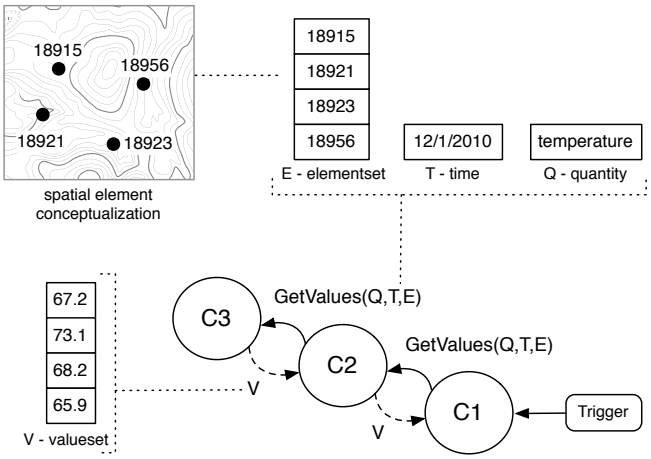


Fig. 1: OpenMI pull-based execution. Solid lines indicate function calls and dashed lines indicate the flow of data.

called a *trigger* begins by calling GetValues on one of the components. When GetValues is called on a component, it executes as many time steps as necessary to advance to the requested point in simulation time and returns a valueset corresponding to that time. Thus a component only executes time steps as-needed to respond to a GetValues call. If it needs input from another component in order to execute a time step it calls GetValues on that component and blocks until a valueset is returned. The components take turns executing synchronously and *pull* data from each other until the simulation completes.

Compositions of linked components can be created and executed using visual software tools. A scientist chooses a set of components, and for each one, assigns each output exchange item to another component's input exchange item. These assignments are called *links* and there may be multiple links between two components and may be in the same or opposite directions.

In this work we present the design and evaluation of a general-purpose Data Collector Component (DCC) that is capable of collecting data from OpenMI components and delivering them to online repositories. We describe the design and implementation of the DCC in the following section and present our experimental results in Section 3. We review related work in Section 4 and present our conclusions in Section 5.

2. Methods

Figure 2 illustrates the movement of data through a distributed data collection system for linked model components. Compositions of linked components execute on cluster nodes. Each composition includes a DCC that collects data from the components and delivers them to web services. Any web service that is capable of accepting data items consisting of a quantity identifier, date, list of location identifiers, and

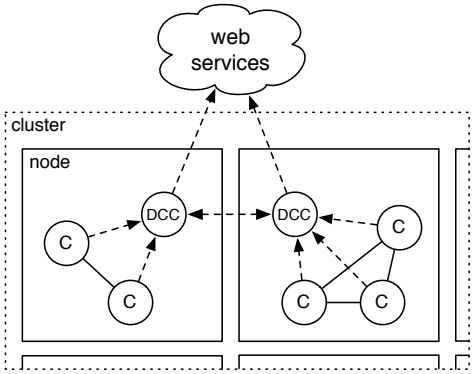


Fig. 2: System overview.

list of values, can be used.

2.1 Web Services

The Open Geospatial Consortium [2] publishes interface standards for location-based information and services to support interoperability. The Geographic Markup Language (GML) [3] standard defines XML schemas for geospatial information including observations data, which is capable of describing model output from OpenMI components. The prototype implementation of the DCC uses this XML schema to represent the data as they are sent to web services. An example of an XML document that conforms to the schema is given in Figure 3 (we used a more succinct gml:id attribute in place of a gml:location element). Additional XML schemas, such as Observations and Measurements [4] could be incorporated into the DCC as well.

```
<gml:Observation gml:id="18951">
  <gml:validTime>
    <gml:TimeInstant>
      <gml:timePosition>2010-12-01T12:00:00</gml:timePosition>
    </gml:TimeInstant>
  </gml:validTime>
  <gml:resultOf>
    <app:Temperature>67.2</app:Temperature>
  </gml:resultOf>
</gml:Observation>
```

Fig. 3: GML description of a single value.

If a DCC were to make a web service call each time it collects a valueset then the execution of the simulation would be paused for the duration of the call due to the synchronous execution of components. In addition, sending a single valueset in each web service call may result in inefficient network utilization when the network latency is comparable to the transmission time of the valueset.

In the ideal case the collection of valuesets would not increase the runtime of a simulation and a sufficient number of valuesets would be transferred in each web service call to achieve efficient network utilization. To these ends the DCC utilizes a buffer that enables the sending of valuesets to be

asynchronous with respect to their collection and allows for the coalescing of multiple valuesets into a single web service call. The buffer is shared among all DCC's across a cluster to provide a total buffer size that is greater than the local buffer size of each individual and to allow cooperation in delivering the buffered data.

The implementation of the DCC consists of a buffering module and a delivery module as illustrated in Figure 4. The buffering module collects valuesets from components and stores them in the shared buffer. The delivery module removes valuesets from the buffer and delivers them to web services. The behavior of these modules is dictated by three

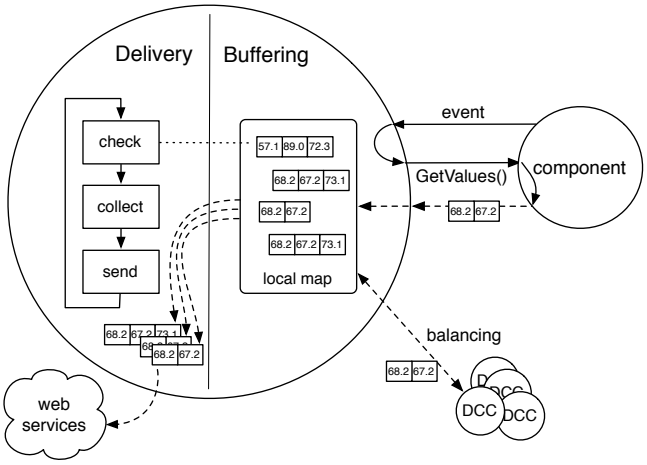


Fig. 4: Operation of the data collector component. Solid lines indicate function calls and dashed lines indicate the flow of data.

parameters: maximum local buffer size, maximum residence time, and minimum delivery size. These are described in the following sections.

2.2 Buffering

A DCC may be linked to one or more components within a composition. Components raise an event each time output data is produced, typically after each time step (the DataChanged event). The DCC listens for this event and in response calls the component's GetValues function to obtain the newly created valueset and places it into the buffer.

The shared buffer is provided by an open source data distribution platform (Hazelcast [5]) that is compiled into, and runs as a set of background threads within, each DCC. It manages a distributed map data structure, dynamically discovering peers via multicast and communicating via TCP/IP. The entries in the map are evenly distributed among all peers and each peer holds a portion of the entries in a local map. Entries are keyed by a universally unique identifier and are variably sized and include the quantity identifier (string), timestamp (long), elementset identifier (string), expiration date (long), valueset data (byte array), and valueset data size (long). The memory overhead of storing an entry in the distributed map data structure is approximately 260 bytes.

The amount of memory dedicated to the local map is dictated by the maximum local buffer size parameter, thus the total memory available to the distributed map is the sum of all peers' local maximums. If the addition of an entry into the buffer would cause the local map's size to become greater that the maximum, then the buffering module waits until there is available space, during which the execution of the simulation is blocked. The buffering module relies on the delivery module to remove entries from the buffer and send them.

2.3 Delivery

The removal of entries from the buffer by the delivery manager is dictated by two parameters: minimum delivery size and maximum residence time. The minimum delivery size provides a means to regulate network efficiency. The delivery manager attempts to remove enough valuesets from the buffer to meet the minimum delivery size before sending them in a single web service call. This may cause entries to remain in the buffer for extended periods of time. This may be acceptable in cases in which the data is being archived, but in cases where the data is consumed as the simulation is being carried out, it may be necessary to place a constraint on the duration that an entry may reside in the buffer before it is delivered. This is controlled by the maximum residence time parameter, which places a limit on the length of time an entry remains in the buffer. Each entry in the buffer has an expiration date that is calculated based on the creation date and maximum residence time. The maximum residence time has higher priority than the minimum delivery size, so in some cases a web service call may contain a small amount of data in order to enforce the time constraint.

The minimum delivery size is specified in terms of the number of values per web service call rather then the number of bytes of serialized XML because different web services may utilize different XML schemas and the latter would require *a priori* knowledge of the serialized XML size for any valueset. Identifying the serialized XML size for a valueset would require either (1) the serialized XML size per value to be known or (2) valuesets to be temporarily serialized to XML as the buffer is being inspected. The former would require calculating the serialized XML size per value for each XML schema and the latter would consume additional system resources.

The following algorithm is used by the delivery manager. The delivery thread periodically iterates over the entries in the local map and determines (1) whether there are any entries that have expired and (2) whether there are enough entries to meet the minimum delivery size. If either case is true, the delivery thread iterates over the local map to identify the entry with the lowest expiration date and removes it. It repeatedly iterates over the entries, removing

the entry with the lowest expiration date, until either (1) enough entries have been collected to meet the minimum delivery size, or (2) the local map is empty. Only the entries in the local map are iterated in order to avoid global operations and improve scalability. The valuesets within the entries are deserialized from their byte array representation, coalesced, serialized into XML, and then sent in a single web service call. The process repeats until both the simulation is completed and the number of entries delivered is equal to or greater than the number of entries inserted into the local buffer. The latter ensures that each DCC delivers a fair share of the entries and that only DCC's with excess capacity deliver more entries then they collect.

When a valueset is delivered to a web service it must include information about the location that each value represents. This information is not stored inside the buffer entries because it would be redundant as the location information is identical for all valuesets that correspond to a common elementset. Elementsets are static during a simulation run so there is typically a high ratio of valuesets to elementsets. The buffer entries only store the elementset's identifier and the actual elementset information is stored in a separate distributed map. In this way a DCC can lookup the complete elementset information for any valueset before it is delivered.

3. Experimental Results

We conducted a performance study using an onsite Linux-based Beowulf cluster. The compute nodes had 2 quad-core 2.3 GHz Opteron 2376 processors with 8 GB of memory and the server node had a quad-core 2.7 Ghz processor and 8 GB of memory. All nodes were connected via gigabit ethernet. The software components were implemented in Java using the Alterra OpenMI 1.4 SDK and the web service was SOAP-based and implemented in PHP hosted by the Apache HTTP server within a Windows virtual machine.

To represent a model component we created a *producer* component that used a fixed-length time step of 1 day and would sleep for a fixed amount of time between time steps to mimic the time spent calculating a time step. On each time step a single valueset was generated and collected by the DCC linked to it.

3.1 Minimum Delivery Size

The minimum delivery size that maximizes throughput to the web service is dependent on several factors including network latency, available bandwidth, and software performance. We conducted a series of measurements to empirically identify the ideal delivery size for our experimental configuration.

To establish a baseline of the expected performance of the DCC, we independently measured the maximum throughput from a benchmark Java application to the Apache/PHP web service and found it to be 47 MB/s. This performance was only possible when the ChunkedStreamingMode of the java.net.HttpURLConnection object was enabled, which prevents the complete POST request from being buffered in memory and streams the data directly from disk to the connection's input stream (although not all web servers support this mode).

We configured a single composition consisting of a producer component linked to a DCC and measured the throughput from the DCC to the web service in a series of simulations. In each simulation the producer generated a number of valuesets that were collected by the DCC and individually sent to the web service in separate calls. The average throughput (over all sends in each simulation) achieved in each simulation is shown in Figure 5.

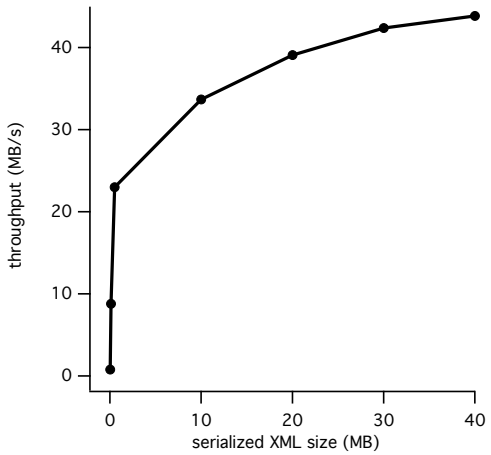


Fig. 5: Impact of minimum delivery size on throughput.

As the data size increased, the throughput increased as well until the maximum throughput was reached, at which point larger data sizes no longer improved the throughput. To achieve at least 50% of the maximum throughput it was necessary to set the minimum delivery size to 2.6×10^5 values which ensures that each web service call contains at least 11 MB of serialized XML data.

3.2 Maximum Residence Time

The maximum residence time imposes a limit on the amount of time that an entry may reside in the buffer. When expired entries are detected in the buffer, all expired entries are removed from the buffer, along with any additional entries necessary to meet the minimum delivery size, and are sent in a single web service call. Thus some entries may be removed from the buffer before they expire resulting in an average residence time that is less than the maximum residence time.

To investigate the effect of the maximum residence time, we measured the average residence time of entries that were added to the buffer at a regular interval. The testing configuration included a single producer component that generated a valueset at a regular interval that was shorter

than the maximum residence time in each case causing several entries to be added to the buffer before one of them expires. The buffer was configured such that entries were only removed as a result of an expiration (unlimited maximum buffer size) and when an expiration occurs all entries were removed from the buffer and sent in a single web service call (minimum delivery size set to maximum).

The results indicate that the average residence time was one-half the maximum residence time in all cases. The expiration of the first entry added to the buffer causes all the entries to be removed and delivered, all of which spent differing amounts of time in the buffer. In general, the sum S of the residence time rt of n entries added to the buffer at a fixed interval i is:

$$\begin{aligned} S_n &= rt_1 + (rt_1 + i) + .. + (rt_1 + (n-1)i) \\ &= n/2(rt_1 + rt_n) \end{aligned} \quad (1)$$

The first entry added to the buffer has the maximum residence time RT_{max} and the last one added has 0 residence time, thus the average residence time RT_{avg} is given by:

$$RT_{avg} = (n/2(0 + RT_{max}))/n = RT_{max}/2 \quad (2)$$

With respect to network efficiency, setting a low maximum residence time resulted in inefficient bandwidth usage because the minimum delivery size was not met. In such cases, network efficiency can be improved by increasing the maximum residence time.

3.3 Buffering

The primary purpose of the buffer is to provide a means for the asynchronous delivery of data to minimize the impact of the data collection on the simulation runtime. To evaluate the buffer's utility in this respect we measured the simulation runtime of a single composition consisting of a producer component linked to a DCC with varying buffer sizes.

The producer generated valuesets of 100 KB at a fixed interval and the DCC had an unlimited maximum residence time and a minimum delivery size equivalent to the size of the valueset so that valuesets were delivered to the web service in individual calls. We imposed a latency on the web service of twice the interval so that data was added to the buffer at exactly twice rate at which it was removed causing the amount of buffered data to increase throughout the simulation.

We measured the *simulation time* as the time spent by the producer executing time steps. In general the minimum buffer size BS_{min} to ensure no impact on runtime is given by:

$$BS_{min} = (InRate - OutRate) \times T_{sim} \quad (3)$$

where $InRate$ is the rate at which data is added to the buffer, $OutRate$ is the rate at which data is removed, and T_{sim} is the simulation time. In this experiment the expected minimum buffer size was (20.0 KB/s - 10.0 KB/s) $\times$ 2500 s = 25.0 MB.

Results are given in Figure 6. Given a sufficient buffer size the runtime remains constant with minimal overhead added by the data collection and sending (adding only the time to retrieve the valueset from the component and insert it into the buffer, both of which occur in memory). The results are consistent with the expected minimum buffer size as speedup becomes constant as the buffer size approaches 24 MB.

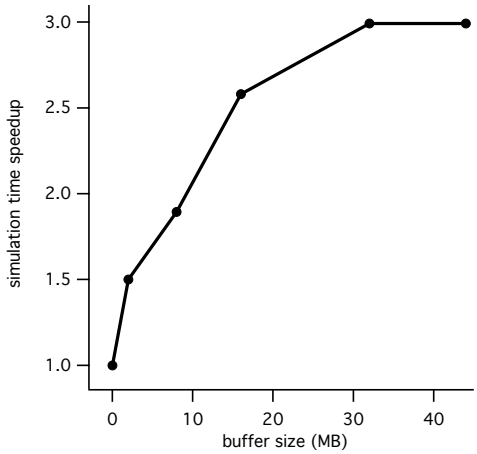


Fig. 6: Speedup for varying buffer sizes.

3.4 Distributed Cooperation

The entries in the local buffer of each DCC are evenly distributed among all the active DCC's on a cluster. This results in entries migrating away from DCC's that are collecting faster then they are delivering, and toward DCC's that are delivering faster then they are collecting, which enables cooperative sending of the data.

To evaluate the effect of cooperation among DCC's we executed concurrent simulations on multiple cluster nodes with differing rates of data collection and data delivery. A single composition consisting of a single DCC and producer executed on each compute node and we measured the *completion time* which we define as the amount of time necessary for all data to be delivered. In the ideal case the completion time is equivalent to the simulation time, but may extend past the simulation time if the simulation completes before all data is delivered.

Four *fast-production* nodes were configured such that data was produced at a faster rate than it could be delivered, and the other *slow-production* nodes were configured such that data was produced at a slower rate than it was delivered. Given a sufficient number of nodes, the runtime remains constant with minimal overhead added by the data collection and sending (adding only the time to retrieve the valueset from the component and insert it into the buffer, both of which occur in memory). The following equation describes the necessary balance of capacity across a cluster to ensure

that simulation runtimes are not affected by the data collection. For N nodes over a given simulation period:

$$0 = \sum_{i=1}^{N} InRate_i - OutRate_i \quad (4)$$

In this configuration, the 4 fast-production nodes inserted 0.5 entries per second while the remaining slow-production nodes each inserted 0.01 entries per second. All nodes removed entries at the rate of 0.1 entries per second. By Equation 4, 32 slow-production nodes would be sufficient to collectively match the rate of 1.6 entries per second inserted by the four fast-production nodes. This assumes that the slow-production nodes are actively delivering data during the complete simulation period.

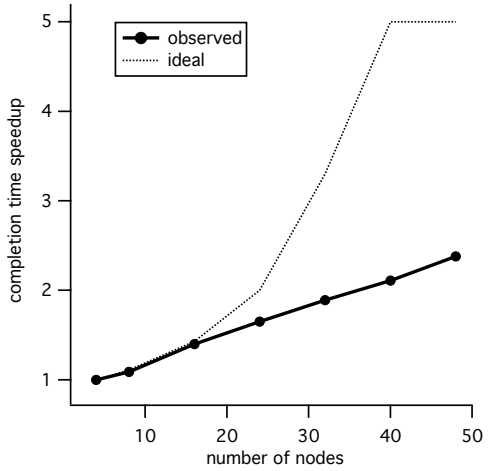


Fig. 7: Speedup for varying numbers of nodes.

We observed near-ideal speedup in the average completion time of the four fast-production nodes for up to 16 nodes as shown in Figure 7. As the number of nodes increased beyond 16, the rate at which entries were added to the distributed buffer was not sufficient to ensure each slow-production node had entries in its local buffer, resulting in the slow-production nodes not sending at all times (as in the ideal case). For example, each slow-production node spent an average of 31% less of its excess time delivering data when there were 48 nodes than when there were 16 nodes. Enabling the cooperation across nodes reduces the completion time, but achieving ideal speedup is contingent on the availability of sufficient data at each node.

4. Related Work

The DCC automates both the task of collecting model output data and transferring them to online services. These are typically performed manually or using custom software [6] that execute general-purpose tools such as GridFTP [7]. Web services have been utilized in modeling and simulation since their conception as both a means to access data and to control the execution of online models [8], [9], [10], [11]. The DCC provides a point of integration between linked models and any Internet-connected data platform that supports web services, including Workflow Management Systems such as Taverna [12] and Vistrails [13] and distributed data storage systems such as iRODS [14] and HIS [15]. It complements existing methods for input data retrieval from online services for OpenMI linked models [16], [17], [18].

5. Conclusions

We presented the design of the Data Collector Component (DCC) for OpenMI components and evaluated its performance. The DCC collects model output data from model components and efficiently delivers them to web services. It utilizes a distributed buffer optimized for the unique behavior and constraints of the OpenMI. General-purpose data collector components simplify the task of collecting model output within a grid environment and facilitate storage and post-processing by online services.

The DCC consists of a buffering module and a delivery module. The buffering module obtains output data from one or more components as they are created and stores them in a distributed buffer that is shared among all DCC's executing on a cluster. The delivery module continuously monitors the buffer and delivers data to web services in a way that balances efficiency and latency.

We evaluated the performance of the DCC and its sensitivity to its three parameters: maximum buffer size, minimum delivery size, and maximum residence time. The minimum delivery size was found to have a significant impact on the throughput and to achieve at least 50% of the maximum throughput each message must contain at least 2.6×10^5 elements in our experimental configuration. The maximum residence time resulted in an average residence time that is one-half the maximum residence time when entries are added to the buffer at a regular interval. We found that buffering resulted in maximum speedup in simulation time (a factor of 3) within a single composition, and distributed cooperation resulted in a speedup in the completion time by a factor of 2.4.

As the importance of data availability, interoperability and transparency continue to rise, so too does the need for software tools to facilitate these. General-purpose tools that intelligently and efficiently collect and deliver data will become an essential part of OpenMI linked models on workstations and grids alike and this work provides a starting point for such tools.

6. Acknowledgments

This work was supported by the National Science Foundation (grants GEO0909515, EPS0919443, EPS1006860). Access to the Beocat compute cluster at the Dept. of Computing and Information Sciences at Kansas State University was appreciated.

References

[1] J. B. Gregersen, P. J. A. Gijsbers, and S. J. P. Westen, "OpenMI: Open modeling interface," *J. Hydroinform.*, vol. 9(3), pp. 175–191, 2007.

[2] OGC, "Open geospatial consortium," 2012, http://www.opengeospatial.org.

[3] C. Portele, "OpenGIS geography markup language (GML) encoding standard," Open Geospatial Consortium, 2007, OGC 07-036.

[4] S. Cox, "Observations and measurements - XML implementation," Open Geospatial Consortium, 2011, OGC 10-025r1.

[5] T. Ozturk, "Scalable data structures for java," in *Devoxx*, Metropolis Antwerp Belgium, November 2010.

[6] M. Papiani, J. L. Wason, A. N. Dunlop, and D. A. Nicole, "A distributed scientific data archive using the web, XML and SQL/MED," in *SIGMOD RECORD*, vol. 28, 1999, pp. 56–62.

[7] G. Toolkit, "GridFTP user's guide," 2012, http://www.globus.org.

[8] S. Chandrasekaran, G. Silver, J. Miller, J. Cardoso, and A. Sheth, "Web service technologies and their synergy with simulation," *Winter Simulation Conference*, vol. 1, pp. 606–615, 2002.

[9] J. M. Pullen, R. Brunton, D. Brutzman, D. Drake, M. Hieb, K. L. Morse, and A. Tolk, "Using web services to integrate heterogeneous simulations in a grid environment," *Future Gener. Comput. Syst.*, vol. 21, pp. 97–106, January 2005.

[10] S. Shasharina, C. Li, R. Pundaleeka, N. Wang, D. Wade-Stein, D. Schissel, and Q. Peng, "HDF5WS – web service for remote access of simulation data," *APS Meeting Abstracts*, p. 2014, October 2006.

[11] J. Horak, A. Orlik, and J. Stromsky, "Web services for distributed and interoperable hydro-information systems," *Hydrol. Earth Syst. Sci.*, vol. 12, pp. 635–644, 2008.

[12] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn, "Taverna: a tool for building and running workflows of services," *Nucleic Acids Research*, vol. 34(Web Server issue), pp. 729–732, 2006.

[13] L. Bavoil, S. P. Callahan, C. E. Scheidegger, H. T. Vo, P. J. Crossno, C. T. Silva, and J. Freire, "Vistrails: Enabling interactive multiple-view visualizations," *Visualization Conference, IEEE*, vol. 0, p. 18, 2005.

[14] A. Rajasekar, M. Wan, R. Moore, and W. Schroeder, "A prototype rule-based distributed data management system," in *HPDC workshop on Next Generation Distributed Data Management*, Paris, France, 2006.

[15] T. Whitenack, "CUASHI HIS Central 1.2," 2010.

[16] T. Bulatewicz and D. Andresen, "Efficient data access for open modeling interface (openmi) components," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) Volume 1, ed. H. R. Arabnia, CSREA Press, Las Vegas, Nevada, USA, July 18-21*, 2011, pp. 822–828.

[17] Q. Harpham, "Future service chain platform," in *First Open Consultation Meeting, Distributed Research Infrastructure For Hydro-Meteorology Study*, Genoa, Italy, October 2010.

[18] KISTERS, "Kisters news," 2010, http://www.kistersnews.com.

N-to-M Mode of IO and Data Management in Numerical Simulations

William W. Dai
Computer, Computational, and Statistical Sciences Division
Los Alamos National Laboratory
Los Alamos, New Mexico, USA

Abstract— *A library for parallel IO and data management has been developed for multi-physics simulations. The goal of the library is to provide sustainable, interoperable, efficient, scalable, and convenient tools for parallel IO and data management for high-level data structures in numerical simulations, and to provide tools for the connection between applications. The library supports the N-to-M mode, in which data on N computer processors are written into M files. The number of files is transparent to users and is chosen by users when files are open. The high-level data structures include those in particle simulations, one- and multi-dimensional arrays, structured meshes, unstructured meshes, the meshes generated through adaptive mesh refinement, and variables associated with these meshes. The IO mechanism can be collective and non-collective. The library is typically used for restarting files, visualization files, and files connecting different applications. The data objects suitable for the library could be either large or small data sets. Even for small data sets, the IO performance is close to the one of MPI-IO performance for large data sets. For resilience, the library guarantees that any parts of files that have been written be readable if computer hardware crashes during writing.*

Keywords: data structure, data management, IO.

1 Introduction

Parallel IO and scientific data management have played an important role since the beginning of large scale scientific computing, and are getting more important due to the increase of the scale of the computing. Existing products, which have partially addressed the issue, include HDF5 [1], SAF [2], CGNS [3], NetCDF [4], Silo [5], UDM [6], and others. Each of the existing products has certain advantages and disadvantages. Some of the products have good functionalities for unstructured meshes, but they either don't have capabilities for running on parallel computer environments or lack for good parallel I/O performance. Some of them are designed for parallel environments, but do not have the capabilities to deal with unstructured meshes, or they get only a fraction of MPI I/O performance. Some have good IO performance but lack the functionality to query data sets for their relationship. Some have a decent IO performance for large data sets, while they failed to deliver the similar performance for small data sets.

The HIO library is for parallel IO and data management for high-level data structures used in numerical simulations. It has been developed under Department of Energy (DoE) Advanced Simulation and Computing (ASC) program for ASC code projects. The HIO library is the further development of the UDM library [6]. The HIO library provides sustainable, interoperable, efficient, scalable, and convenient tools for parallel IO and data management for high level data structures in applications. In the DoE community, such as national laboratories, data files generated in one code project often have to be used in another code project as inputs. The HIO library provides a parallel tool for such connections.

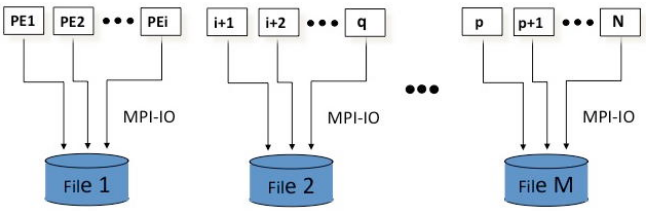


Figure 1. Data on N computer processors are written into M files.

The HIO library writes simulation data on N computer processors to M files on parallel computer environments, i.e., N to M, as shown in Fig.1. The number of files, M, is chosen by users when the files are opened, and M could be one. The library consists of functionalities for IO and data management for high-level data structures encountered in numerical simulations on parallel computer environments, such as data in particle simulations, structured meshes, unstructured meshes, meshes generated from adaptive mesh refinement (AMR), and their associated variables defined on cell centers, vertices, edges, or faces of meshes. The library is built on the top of MPI I/O, and its I/O performance is almost the same as MPI I/O. To our knowledge, the functionality and performance of the library are superior to

existing products for these data structures in numerical simulations.

In this paper, we will report the library, and its main usages. In Section II, we will present the main functionalities of the library. The usage of the library will be demonstrated in Section III. The IO performance of the library will be discussed in Section IV, and in the final section we will discuss resilience of the library and some of our future plans.

2 Functionalties

The HIO library provides IO and data management tools for data in particle simulations, single and multi-dimensional arrays, structured meshes, unstructured meshes, and the mesh generated through adaptive mesh refinement (AMR), and their associated variables defined on these meshes in numerical simulations on parallel computer environments. It also provides a hierarchical data structure within a data file. The files generated through the library are self-described.

2.1 N-to-M Mode

In the current and future computer system, the number of computer processors in a computer system is so large that writing to a single file for a writing event will not be optimal for IO performance and network communication. Therefore, the data on N computer processors, on which a simulation is run, could be written to M files, M could be one, and M could be N although the case with M equal to N is not our focus. For a given N processors, the optimal M depends on individual computer system and software on it, it also depends on the size of the data to be written into this files. For this reason, through the library, M is a file-specific parameter, and some files could be written in the N-to-M mode while others are done through the N-to-1 mode. For this library, this M is a parameter that users can choose when they create files, and by default it is one.

For writing, the library divides N processors into M groups for any M provided by users. For example, a user is to write a file with a file name foo and M = 2. The library will actually create two files with names foo.m00001 and foo.m00002. These two files are also what the user sees, and they are transparent to users. From the names of these files, foo.m00001 and foo.m00002, the user can not see each file contains the data of which group of processors, but this information could be easily queried from a function call of the library.

After the file is created, for example, foo, the library remembers the partition of M files from N processors until the file is closed. Before the file is closed, any subsequent writing into the file, foo, will be automatically partitioned into the M files. The writing call is the same as that for the case with N equal to one, without the parameter M involved.

For the convenience to query data in the M files, each of these M files contains the description of the partition of N processors among the M files and the description of data on all N processors, such as the sizes of data on each processor and the processors writing to any particular file. Each of the M files contains also the information of structure of the files and the description of all the objects in the M files, such as the information written into a single file. Therefore, from each of the M files, users can get the description of all M files.

2.2 Unstructured Meshes

One of the important and powerful functionality in the HIO library is the management of unstructured meshes and their associated variables. The library supports a broad range of unstructured meshes, which include meshes with fixed shapes, arbitrary polygons, and arbitrary polyhedrons. The mesh elements with a fixed shape may be triangles, quadrangles, pentagons, tetrahedrons, pyramids, wedges, pentagon prisms, and points in particle simulations. A mesh element may be a zone, or face, or edge, i.e., a mesh may be a zone-mesh, face-mesh, edge-mesh, and points. An edge-mesh may be one-, or two-, or three-dimensional, and a face-mesh may be either two- or three-dimensional. Mesh elements of a zone-mesh may be made directly from nodes, or the elements may be made from edges, or the elements may be made from faces and the faces are then made from either edges or nodes. The HIO library also supports ghost mesh elements, boundary faces, boundary edges, boundary nodes, slip faces, slip edges, slip nodes, etc. The variables associated with unstructured meshes may be node-variables, or edge-variables, or face-variables, or zone-variables, and variables may be scalars, or vectors, or tensors.

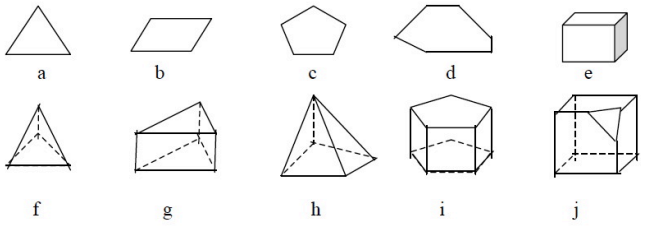


Figure 2. Examples of mesh elements supported in the HIO library for unstructured meshes.

To illustrate the meshes supported in the library, in Fig.2 we list some examples for mesh elements. They include (a) triangles, (b) quadrangles, (c) pentagons, (d) arbitrary polygons, (e) hexahedrons, (f) tetrahedrons, (g) wedges, (h)

pyramids, (i) pentagon-prisms, and (j) arbitrary polyhedrons. The mesh elements may be made from any lower level entities, such as faces, edges, and nodes. Figure 3 shows three possible ways to make up three-dimensional elements.

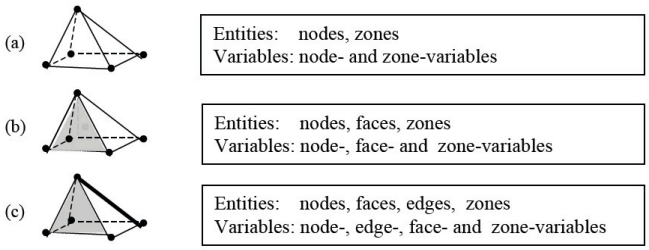


Figure 3. An element may be made from (a) nodes, or (b) faces and nodes, or (c) faces, edges, and nodes.

Although the HIO library covers a broad range of unstructured meshes, a user only has to set up his/her own mesh definition, and all other mesh definitions are hidden from the user. For example, for an unstructured zone-mesh made from nodes, only the list of nodes for each element is needed, if the elements are of a fixed shape, such as prisms. If mesh elements are arbitrary polyhedrons made from nodes, two arrays are needed, one for the numbers of nodes for elements, and the other for the list of nodes for each element. Like the capability for structured meshes, the association between a mesh and a set of variables is automatically built into the library.

2.3 Adaptive Mesh Refinement

Although a mesh generated through AMR may be considered as an unstructured mesh in IO, but it involve unnessary memory copies and additional working memory requrement. The HIO library is able to handle AMR meshs naturally without additional memory.

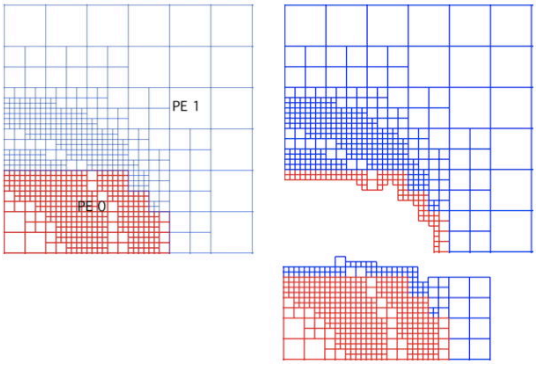


Figure 4. Illustration of ghost cells in cell-based AMR. The left image is the partition of between two processors, and the right one is the part of the mesh on each processor with ghost cells.

For element-based AMR structured meshes in the HIO library, we store the center and width of each element in each dimension. The scalar variables associated with the meshes are one-dimensional and the associated vector variables are two-dimensional arrays. The association between the variables and a mesh is automatically built and stored in the file. For block- or patch-based AMR structured meshes, each block or patch is considered as a standard structured mesh.

The library also support ghost cells in AMR meshes. Some post data anaylizers need variables on ghost cells to derive information around interfaces between processor interfaces, such as a visualization tool calculating isosurfaces. Figure 4 shows the illustration of ghost cells surrounding the part of an AMR mesh on each processor.

2.4 Writing/Reading Descriptions

Users can add any description to any object, such as a file itself, or an array, or a mesh, or a variable, as long as the description is not of the problem-size, and each processor has the same description. More importantly, writing all descriptions almost doesn't have any IO cost, since all the descriptions will be buffered together with all the meta data and are written at the end of a file when the file is closed.

The number of the descriptions and each description can be automatically queried. As writing the description, reading any description does not involve any additional IO cost since all the descriptions together with all the meta data are read into the memory when a file is open.

2.5 Small Data Sets

Writing small data sets into a file on a parallel environment will typically result in very IO performance. The HIO library provides an automatic buffering mechanism so that a large number of small data sets will automatically buffered together before they, together with their names and descriptions, are written into a file. Writing small data sets, users will get the same IO performance as they get for large data sets. But, users don't have to keep track of the locations of each individual small data set in the combined buffer and the disk file.

To read a small data set, the HIO library actually only copies small data set from a buffer to the user's memory. If the buffer is not available yet, the library will automatically read the buffer first, and then copy the data. Therefore, the library doesn't involve reading from a disk with a small set of data.

To users, all the tedious operations necessary for writing and reading the small data sets are behind the scene. Writing

small data sets is the same as writing big data sets, and even the names and arguments of the functions to be called are the same.

2.6 Querying

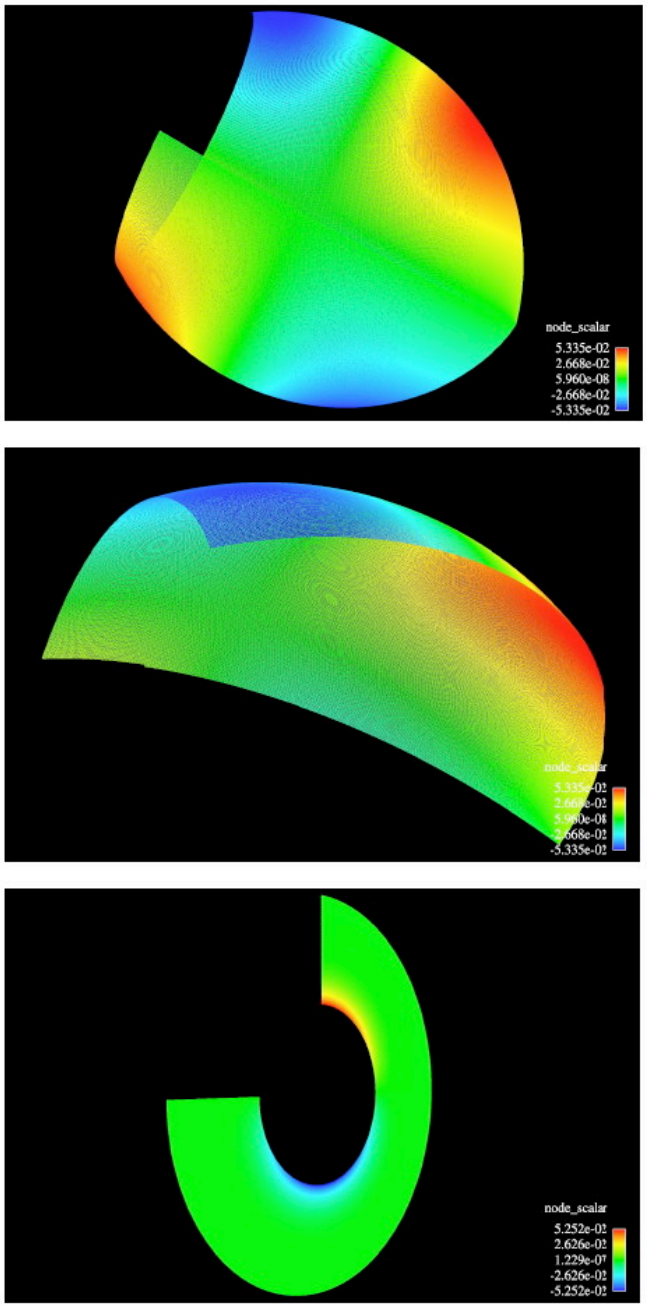


Figure 5. Three parts of an unstructured mesh with 1.6 billion elements. The left is read through an original processor rank, the middle one is read through a set of global element ids, and the right is read through a space domain.

A file written through the HIO library is self-described. All the information in the file may be queried by function calls of the library. For example, for a given file, users may find the number of arrays, meshes, and variables, the description of each array, mesh, and variable, and any association between meshes and variables. Through the querying function in the library, meshes and variables may be directly viewed through parallel graphics tools.

After a data object, such as array, mesh, and variable, is written into a file, users may read any part of the data object in terms of, for example, global ids, or a processor rank, or a space domain. Figure 5 illustrates the capability for reading three parts of an unstructured mesh with 1.6 billion elements. The top image is a part read through a processor rank, the middle one is a part identified through a set of global element ids, and the lower image is read based on a space domain.

As stated beore, one of usages of the library is to connect data files to visualization tools. All the images presented in this paper are produced through the visualization tool Ensight except the the one in Fig.6, which is produced through VisIt. Our simulations generate a set of files for visualtion, and the set of files could be read by either Ensight and VisIt without direct transformation.

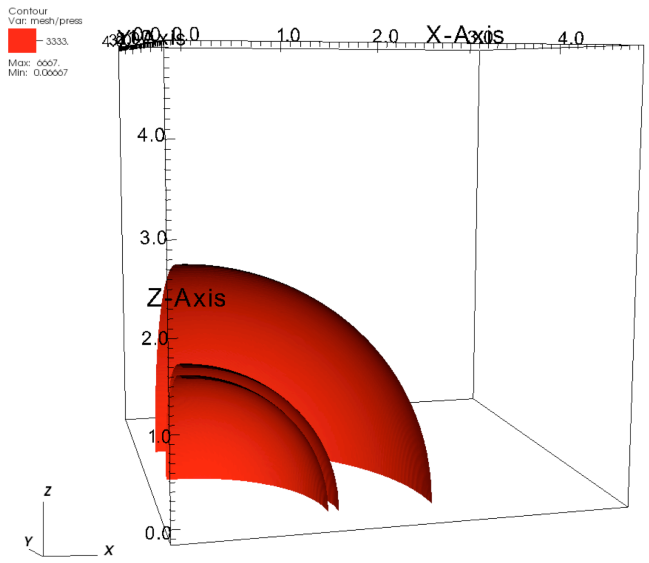


Figure 6. Iso-surfaces of pressure of a three-dimensional simulation.

2.7 Hierarchical Data Format

The HIO library supports the hierarchical data structure within each file, which is equivalent to the Unix file system, although it is not necessary for the functionalities of the library mentioned above. After a file is created, users may create any number of groups within the file. A group is a container in which other groups and data sets may be

created and written. A file is also a group. A data set is an array, or a mesh, or a variable. A number of attributes may be attached to a group or data set. An attribute is any additional description users want to store into the file for a group or data set. Due to the hierarchical data structure and the attributes, users may build their own data format that is self-described.

All the data needed for the hierarchical data structure and attributes are stored at the end of each file, which is written to a file only when the file is closed. Therefore, the cost for the hierarchical structure and attributes is very minimal.

3 Basic Functions and Uages

One of the design principles of the HIO library was a small number of functions. The following is the list of main functions of the library.

- hio_set_mcomm(mfiles)
- hio_open(filename, mode, fid)
- hio_close(fid)
- hio_init(type, fileid, obj)
- hio_write(type, fileid, obj)
- hio_query(type, fileid, filter, nobjs, objs)
- hio_clean(type, nobjs, objs)
- hio_get_size(type, domain, fileid, obj)
- hio_read(type, domain, fileid, objs, nobjs)
- hio_init_append(type, obj)
- hio_finalize_append(type, id)

The first two functions are for opening and closing files. The function hio_init is to initialize any object before it is being used, and the object includes array, structured and unstructured mesh, AMR mesh, and variable defined on a mesh. The function hio_query is for querying, which include querying files, querying variables, querying relationship between variables and meshes, querying attributes, etc. The function hio_clean is to release the memory allocated in the call of the function hio_query. The function hio_get_size to determine the sizes of grid zones, faces, edges, and nodes for a given part of a mesh, for example, a spatial domain, a part associated with a specific processor, or a number of elements. The function hio_read is to read attributes and any data for a given part of a mesh, which include coordinate, mesh, variable, etc. The functions, hio_init_append and hio_finalize_append, together with hio_write are for generating large meshes with a small number of computer processors.

The following is an example to write an unstructured mesh with general polyhedrons. To specify the mesh, each computer processor has a number of elements, nzone. The set of elements have a number of faces and nodes, nface and nnode. The arrays, num_faces_for_zone and facelist_for_zone, are to specify the elements, and the arrays, num_nodes_for_face and nodelist_for_face, are to define the faces. The arrays, x, y, and z, are the locations of these nnode nodes. All these arrays and sizes are local to the processor. Then code to write this unstructured mesh is as follows.

```
hio_unstructured_mesh m;
hio_coord *c = &m->coord;
hio_init(hio_umesh, -1, &m);
m.dims = 3;
m.type = hio_general_mesh;
m.sizes[0] = nzone;
m.sizes[1] = nface;
m.sizes[3] = nnode;
m.num_nodes_for_face = num_nodes_for_face;
m.nodelist_for_face  = nodelist_for_face;
m.num_faces_for_zone = num_faces_for_zone;
m.facelist_for_zone  = facelist_for_zone;
c->coord[0] = x; c->coord[1] = y; c->coord[2] = z;
c->datatype = hio_double;
m.datatype = hio_int;
hio_write(hio_umesh, fileid, &m);
```

After this mesh is written to a file, this mesh can be queried as described before.

The following segment of codes demonstrates the usage to write a large mesh into a file through a small number of processors.

```
hio_unstructured_mesh m;
hio_coord *c = &(m.coord);
hio_init_append(hio_umesh, -1, &m);
m.dims = 3;
m.type = hio_general_mesh;
m.datatype = hio_int;
c->datatype = hio_double;
while (more_block) {
    generate a part of mesh
    write the part to mesh
}
hio_finalize_append(hio_umesh, m.id);
```

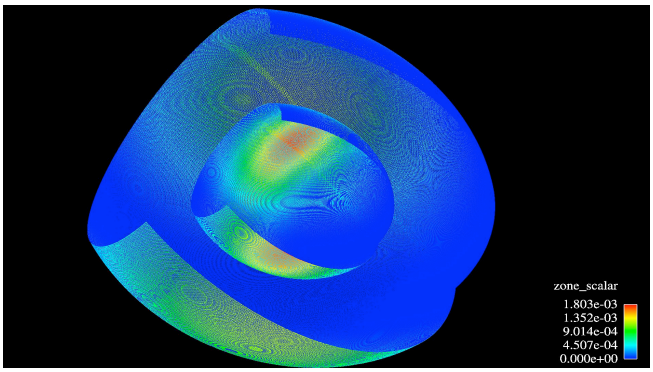


Figure 7. An unstructured mesh with 1.6 billion elements written through the "append" capability in the HIO library.

An example mesh with 1.6 billion of unstructured elements generated through 16 processors in this way is shown in Fig.6.

A cell_based AMR structured mesh is defined through arrays for the centers of elements, x, y, z, and arrays for widths of the elements, dx, dy, and dz. The following segment of codes shows the usage to write a cell_based AMR mesh. An example mesh is shown in Fig.7 that contain four materials.

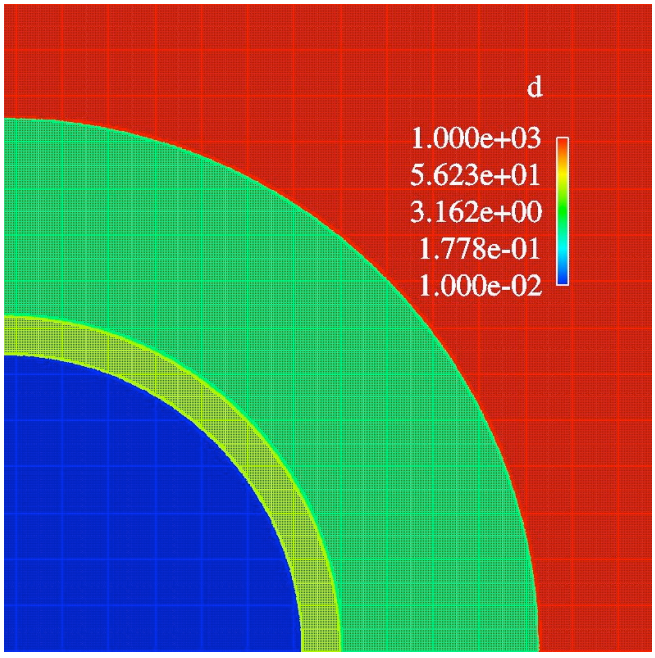


Figure 8. A cell_based AMR structured mesh with four materials and the variable of density written through the HIO library.

```
hio_Structured_Element_AMR m;
hio_init(hio_smesh_element_amr, -1, &m);
m.name = meshname;
m.dims  = 2;
m.datatype_coord = hio_double;
m.size          = nelement;
m.coord[1]   = x;
m.coord[0]   = y;
m.dcoord[1] = dx;
m.dcoord[0] = dy;
hio_write(hio_smesh_cell_amr, fileid, &m);
```

After a mesh is written into a file, a set of variables can be written into the file, and the relationship between the mesh and variables is automatically built. The following codes show the usage to write a scalar variable defined on elements, zone_density, and a vector defined on nodes, node_velocity_x and node_velcity_y.

```
hio_Mesh_Var   var;
hio_init(hio_mesh_var, -1, &var);
var.name          = varname1;
var.mesh_ids[0] = m.id;
var.type          = hio_zone;
var.datatype      = hio_double;
var.rank          = 0;
var.comps[0].buffer = zone_density;
hio_write(hio_mesh_var, fileid, &var);

hio_init(hio_mesh_var, -1, &var);
var.name          = varname2;
var.mesh_ids[0] = m.id;
var.type          = hio_node;
var.datatype      = hio_double;
var.rank          = 1;
var.comp_sizes[0]   = 2;
var.comps[0].buffer = node_velocity_x;
var.comps[1].buffer = node_velocity_y;
hio_write(hio_mesh_var, fileid, &var);
```

After a mesh and a set of variables are written into a set of files. Any part of the mesh and the variables associated with this part of the mesh may be easily read. The following segment of codes shows the usage to read a part of mesh defined through domain, and nvar variables, vars, associated with this domain.

```
int                    nvar;
hio_Domain             domain;
hio_Unstructured_Mesh  m;
hio_Mesh_Var           *vars;
specify mesh and domain
hio_get_size(type, domain, fileid, &m);
allocate space for the part of mesh, and variables
hio_read(hio_umesh, domain, fileid, &m, 1);
hio_read(hio_mesh_var, domain, fileid, &vars,
nvar);
```

The other examples include patch-based AMR structured meshes shown in Fig.8. The patches of the first and second levels of a patch-based AMR mesh in a three dimensional simulation are displayed in the figure. The rectangular with each color in Fig.8 is a patch.

4 Performance

The HIO library is built directly on the top of MPI-IO, and files generated are machine-independent. Its functionality and performance have been tested on from a couple of dozen processors to full machines, and its performance is around 97% of that of the MPI-IO.

The library depends on MPI I/O for its I/O performance, and it currently supports both collective and non-collective writes. The library doesn't explicitly move data between processors. The library itself doesn't directly interact with file systems. If it is necessary, the library have appropriate functions to set parameters of the file system through MPI

calls. If the MPI is tuned to be of high performance in a machine, the HIO library will have high IO performance too.

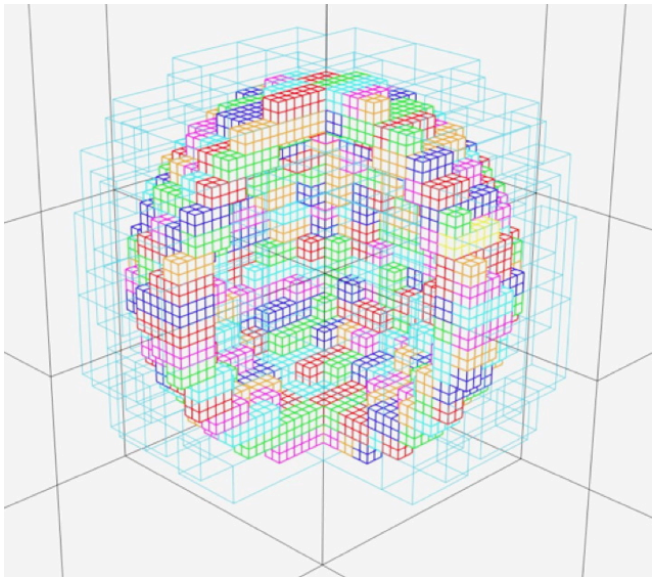


Figure 9. A patch-based AMR structured mesh written through the HIO library. The image the second level of patches on the the top of the first level patches.

To illustrate I/O performance, we used the Q, Lightning, and Lobo machines in the Los Alamos National Laboratory as examples. The Q machine has a HP proprietary parallel file system, and the Lightning and Lobo machines have global parallel file systems provided by Panasas. The MPI library, mpich, is used on the Q and Lighting machines for the tests, and OpenMPI is used on the Lobo machine. An unstructured mesh and its 50 associated variables are used in the example. The mesh is defined by three arrays for coordinates, and an array for the list of nodes for elements. We use 510 processors on the Lightning machine to write the mesh with total elements 1.6 billion and 50 associated variables. The size resulting files is about 405 Gbytes. As stated before, the files generated through the library are slightly larger than the files generated through MPI I/O. This is due to the meta data used in the HIO library. The HIO library gets 97% of writing performance of MPI I/O. On the Q machine, we use 256 processors, the size of resulting files is about 203 Gbytes, and the writing performance is also 97% of the one obtained through MPI I/O. For 1024 processors on the Lobo machine, for a data file of 813 Gbytes, the performance of the HIO library is 98% of the one of raw MPI-IO calls.

A few points about the test are worth being mentioned here. Firstly, for the pure MPI I/O test, all processors collectively write 53 large arrays, and the sizes of data on each processor are roughly equal. Secondly, the file generated through the HIO library is self-described, and may be queried and visualized, but the file generated through MPI I/O may not. Last, for the case with 510 processors, the total overhead of file size in the HIO file is 110 kbytes, or 0.00004%. Each processor contributes about 200 bytes of the 110 kbytes, and remaining 8000 bytes are the overhead for the description of file structures.

For the best possible performance, we take three main steps. We first make sure that the data on each processor are contiguously written onto a disk file, and therefore there is no movement of data during writing. Second, the library collects all the data for descriptions of arrays, meshes, variables, relationship and associations, the hierarchical file structure, etc., and writes the collection only when a file is closed. Third, when reading files, the library reads all the meta data and the file structure together and reads it only once.

5 Acknowledgments

The work presented here has been supported by Department of Energy through the Advanced Simulation and Computing program. The author is grateful to Paul Weber for his effort in readers of visualization tools for the data files written through HIO library.

6 References

[1] Brown, S., Folk, M., Goucher, G., Rew, R., "Software for Portable Scientific Data Management", Computers in Physics, vol. 7, no. 3, pp.304-308 (1993).

[2] Miller, M. C., Reus, J. F., Matzke, R. P., Arrighi, W. J., Schoof. L. A., Hitt, R. T., Espen, P. K., "Enable Integration of High Performance, Scientific Computing Applications: Modeling Scientific Data with the Sets and Fields (SAF) Modeling System", in Computational Science- ICCS 2001, Alexandrov et al. (Eds.), Springer-Verlag Berlin Heidelberg 2001, pp.158-167, 2001.

[3] Poirier, D., Allmaras, S., McCarthy D. R., Smith M., and Enomoto F., " The CGNS System", 39th AIAA Fluid Dynamics Conference, AIAA-98-3007, Albuquerque, NM, June, 1998.

[4] Rew, R. K., Davis, G., "NetCDF: An Interface for Scientific Data Access", IEEE Computer Graphics and Applications, vol.4, pp 72-82, July (1990).

[5] Roberts, L., J., "Silo User's Guide", University of California Research Lab Report, Lawrence Livermore National Laboratory, UCRL-MA-118751-REV-1 (2000).

[6] Dai, W., Aulwes, R., Gaeta, M., and Pfaff, R., Unified Data Model (UDM): A Library for Parallel IO and Data Management, The proceedings of the 2007 International Conference on Parallel amd Distributed Processing Techniquies and Appliciation,Arabnia et al (Eds.), CSREA Press 2007, Vol.II, pp.697-702, 2007.

Scalable Solution of Radiative Heat Transfer Problems by the Photon Monte Carlo Algorithm on Hybrid Computing Architectures

Joo Hong Lee, Mark T. Jones and Paul E. Plassmann
Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, Virginia 24061

Abstract – The simulation of Radiative Heat Transfer (RHT) effects by the Photon Monte Carlo (PMC) method is a computationally demanding problem. In this paper we present results and analysis of a new algorithm designed to solve this problem on a hybrid computing architecture. This architecture includes distributed memory, shared memory, and Graphics Processing Unit (GPU) accelerated components. In this paper we present an approach to obtain good parallel performance based on a partitioning of the application software into two parts. The first part is a multi-threaded application code that manages the ray tracing aspects of the PMC. The second part is an asynchronous, GPU-accelerated pseudo-random number generation library. An advantage of this approach is that this software framework can be easily translated to other Monte Carlo applications. We present experimental results from a large-scale hybrid computer for a standard RHT model problem and compare these results to our analytical model.

Keywords: Monte Carlo algorithms, radiative heat transfer, GPU acceleration, hybrid computing, scientific computing.

1. Introduction

Radiative Heat Transfer (RHT) plays a central role in many important engineering applications that involve combustion. The computational cost of modeling RHT effects accurately can be extremely high due to its highly nonlinear and non-local nature [1]. This nonlinearity arises because RHT rates typically depend on the fourth power of the temperature [2]. Thus, applications that involve the computation of these rates, such as with combustion, are highly sensitive to the accuracy of these temperature calculations. Omitting RHT effects from simulations in such applications can lead to inaccurately computed temperature profiles, which in turn affects the stability and the accuracy of the calculation of other variables [3].

The non-local nature of RHT comes from the fact that the photons that carry radiation (and energy) can be absorbed far from the physical position that they are emitted. Because of these non-local effects, conservation laws cannot be applied over an infinitesimal volume, but instead must be applied over the entire computational domain. The Photon Monte Carlo (PMC) method can be effectively used in the solution of thermal radiation problems [4]. This method is based on a model of radiative energy traveling in discrete packets (like photons) and the computation of the effect of these photons while traversing, scattering and interacting with matter within the computational domain. Advantages of this method include: an ability to deal with complex geometries; an ability to handle non-uniform temperature fields; the ability to include photon scattering; and the ability to employ a great variety of methods to include specialized radiative properties of the enclosure or the transport domain [5].

This paper is organized as follows. In section 2 we describe the algorithm of PMC method and our software framework. In section 3 we present an analysis of the performance of this proposed approach on a hybrid computing architecture. In section 4 we compare experimental results of the performance of a representative PMC simulation with this analytical model. We present our conclusions in section 5.

2. The Photon Monte Carlo Method

The Photon Monte Carlo (PMC) method is a sampling method based on simulating the movement and absorption of photon bundles (rays) through a discretized computational domain. The advantage of this approach, as opposed to other RHT approximation schemes, is that its overall computational cost grows slowly as a function of the complexity of the RHT problems [4]. An additional advantage is that increased accuracy can be obtained by using larger numbers of photon bundles. Hence, the PMC method is well suited to radiation calculations that include complex geometries, non-trivial absorption properties, and singular effects such as scattering. In this section we describe the basic PMC algorithm. We also review how pseudo-random numbers are used within the PMC algorithm and ultimately in the overall RHT simulation.

2.1. The PMC Algorithm

In numerical simulations that include a RHT component, the computational domain contains a participating medium (a material that both emits and absorbs photons). The PMC method traces a statistically significant sample of photon bundles from their point of emission within the medium to

a point of absorption within the medium or its boundary. When the photon bundle is absorbed, its energy is added to the local energy of the absorbing element within the discretized medium. With this approach, the PMC method is able to calculate the energy gain or loss for every element within the computational domain.

The tracking of the photon bundles through the computational domain requires that the PMC algorithm model several types of element interactions. These interaction types are illustrated in Figure 1. On the left, Figure 1 (a), we show a photon bundle entering an element and being absorbed within the element. In the middle, Figure 1 (b), we show a photon bundle entering an element and being scattered off of a particle within the element. On the right, Figure 1 (c), we show a photon entering an element, traversing the entire element, and exiting the element to enter a neighboring element. For the computational results presented in this paper we employ a software framework capable of modeling these element interactions and tracing the photon bundles through a computational mesh [6].

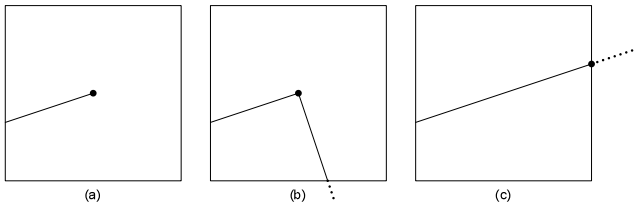


Figure 1. Possible photon interactions within an element. In (a) we show the photon bundle being absorbed within the element, in (b) the photon is scattered off a particle within the element, and in (c) the photon bundle is transmitted through the element.

The PMC algorithm can be used to solve a wide range of RHT problems. In this paper, we use the algorithm to solve a simple model problem. The model problem we use is a three-dimensional rectangular solid domain with two plates with different temperatures on opposite walls and periodic boundary conditions in the other two orthogonal dimensions [4]. The RHT problem that we solve involves computing the temperature as a function of position between the two plates. For a fixed computational mesh the overall computational work grows linearly with the number of photon bundles that we track through the domain. By increasing the number of photon bundles used in the simulation, we can examine the "scaled" speedup of the simulation when run on a hybrid parallel computer architecture. In addition, as we increase the number of photons, the accuracy of the computed solution improves and we can verify the statistical independence of sampling done by the photon bundles by looking at the convergence of the temperature from independent samples.

2.2. Pseudo-Random Numbers and PMC

For every photon bundle, the PMC algorithm must determine a point of emission, a direction of emission, a wavelength, a point of absorption, and various other properties that are independently chosen from probability distributions. Because of the large number of required pseudo-random numbers, a profile of the PMC code running a standard CPU shows that 90% or more of the computational time is spent generating these numbers. This percentage can be even higher as the complexity of the radiative properties and the accuracy required from the simulation increases. Based on this observation, a more efficient scheme for generating these pseudo-random numbers can dramatically improve the overall performance of the PMC algorithm. In the following section, we present a GPU accelerated pseudo-random number generation algorithm. In addition, we present an analysis of the running time of this algorithm and show how it achieves much of this potential efficiency gain for the PMC algorithm.

3. A Theoretical Analysis of the GPU Accelerated Algorithm

As discussed in the proceeding section, the PMC method for solving RHT problems involves the tracing of large numbers of statistically independent photon rays [4]. The statistical independence and the spatial integration of the effect of the photons as they traverse elements with non-trivial absorption properties require the use of large numbers of pseudo-random numbers [4]. The overall PMC software framework that we use for the experiment results presented in this paper was developed in [6]. The pseudo-random number generation function calls in this code are replaced with calls to the GPU accelerated software library developed in [7].

In this section we present an analysis of the running time of the PMC code on a hybrid computing architecture. This analysis consists of two parts—first an analysis of the pseudo-random number generation and, second, an analysis of the multi-threaded PMC calculation that makes use of these pseudo-random numbers.

3.1. Pseudo-Random Number Generation

The approach used to generate pseudo-random numbers for use in multi-threaded Monte Carlo applications is presented in detail in [7]. We review the analysis developed in [7] in order to be able to derive an overall analysis of the PMC simulation.

As discussed in [7], using the GPU to generate pseudo-random numbers involves three main factors: the transfer of state tables from CPU to GPU memory, the actual computation of the pseudo-random numbers on the GPU stream processors, and finally the copying back of the state tables and the pseudo-random numbers from the GPU to CPU memory.

The memory transfer time between the CPU and GPU and back again can be modeled by a linear dependence with respect to the amount of data transferred. Accordingly, if we denote the time required to copy m bytes of data from the CPU to the GPU by $T_{CPU\rightarrow GPU}(m)$ and the time required to copy m bytes of data from the GPU to the CPU by $T_{GPU\rightarrow CPU}(m)$, we have the linear relations

$$T_{CPU\rightarrow GPU}(m) = t_{CG}\, m + t_s \qquad (1)$$
$$T_{GPU\rightarrow CPU}(m) = t_{GC}\, m + t_s.$$

In these formulae t_s is a "start up" time for the copy, t_{CG} is the incremental time required to copy each addition byte of data from the CPU to the GPU, and t_{GC} is the incremental time required to copy each addition byte of data from the GPU to the CPU. These constants are architecture dependent and can easily be measured. For example, for the Athena system used for the results presented in the experimental section of this paper we obtained the data shown in Figure 2.

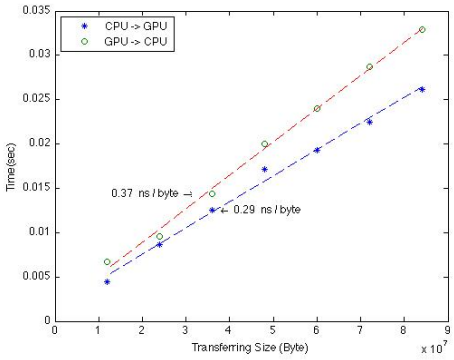


Figure 2. Experimental results from the Athena system showing the time (in seconds) required transferring data between the CPU and the GPU (and visa versa) as a function of the number of bytes transferred. Note the different incremental transfer rates to and from the GPU.

Using a linear least squares fit to the data shown in Figure 2, we obtain the values for the constants in (1) as shown in Table 1.

Table 1. The constants t_s, t_{CG} *and* t_{GC} obtained by a linear least-squares fit to the data in Figure 2. These constants are for the Athena system, the hybrid computing system used for the experimental results.

Constant	Time
t_s	*3.0ms/copy*
t_{CG}	0.29*ns/byte*
t_{GC}	0.37*ns/byte*

The second aspect of computing the pseudo-random numbers on the GPU is the time to execute the "kernel" (the OpenCL program that contains the instructions that are executed on the GPU). This time can be modeled as consisting of two parts, a "kernel start up time" T_s and a "kernel execute time" which we denote by T_e. The total time to execute the kernel, T_k, is modeled as the sum of these two terms as

$$T_k = T_e + T_s, \qquad (2)$$

The pseudo-random number generator works by generating a sequence of numbers in a loop—we denote the number of times through the loop, or the "kernel cycle," as n_k. Empirically we determine that T_s can be modeled as a function of n_k by the equation

$$T_s(n_k) = an_k + b, \qquad (3)$$

where a is an incremental rate measured to be 100*ns/kernel-cyle*, and b is a fixed setup time measured to be 1*ms*. Each kernel cycle, the GPU tries to schedule some number of threads, n_{wgs}, called the "working group size" on each compute unit in the GPU. The GPU performance is, however, limited by the number of stream processors that it has per compute unit. We denote this number by p_{wgs} (this number is 32 for the Athena system). Given that it takes some amount of time to execute the thread, say t_c^{GPU}, then the second term, the "kernel execute time," can be modeled as

$$T_e(n_k, n_{wgs}) = n_k\, t_c^{GPU} \lceil n_{wgs}/p_{wgs} \rceil, \qquad (4)$$

where t_c^{GPU} was measured to be 400*ns/number* for the Athena system. Using the equations (3) and (4) to model the overall execution time for one kernel cycle as given in (2), we obtain the black '*' points shown in Figure 3. In this figure the number of kernel cycles, n_k, is fixed at 10,000; we then measure the time it takes for the kernel to execute. The measured times are shown as the green '+' symbols in this graph. As the work group size increases beyond multiples of 32 (e.g., 32, 64, and 96), we observe discrete jumps in the measured times as predicted by equation (4).

A good way to parameterize the performance of the pseudo-random generator is in terms of the number of work items, n_{wi}, which is the product of the number of work groups, n_{wg}, and the work group size, n_{wgs}, as follows

$$n_{wi} = n_{wg}\, n_{wgs}. \qquad (5)$$

This a good way to parameterize the scaling of the parallel algorithm because the number of work items, n_{wi}, represents the number of independent "tasks" that are to be executed on the GPU. However, there are two limitations to the number of these tasks that can be executed in parallel. First, only one work group can use a compute unit at a time; hence, the number of work groups that can execute in parallel is limited by the number of compute units on the GPU. We denote the number of compute units on the GPU by n_{CU}. Second, as discussed above, the number of stream processors per compute unit, p_{wgs}, limits the number of threads that can execute at one time on a compute unit.

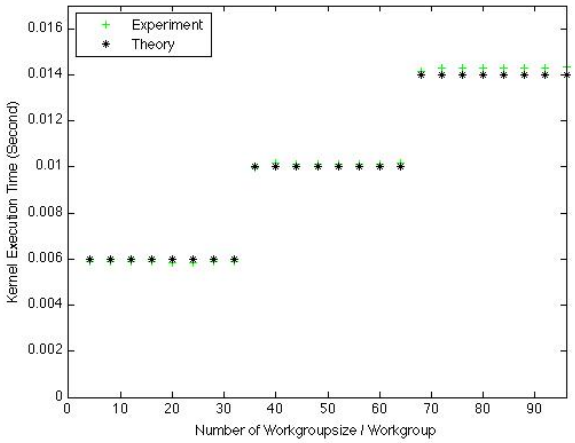


Figure 3. The time measured for the kernel to execute as a function of the work group size per work group (or compute unit). For this data we fixed the number of work groups to be one. The experimentally measured data from the Athena system is shown as the green '+' points, the modeled times, based on equation (2), are shown as the black '*' points on this graph. In this figure the number of kernel cycles is fixed at 10,000.

A complete model of the time required to generate the pseudo-random numbers requires that we also include the time necessary to copy the pseudo-random number seed tables back and forth between the CPU and GPU memory (one table for each work item) and to copy the pseudo-random numbers from the GPU to the CPU memory. To help amortize the cost of copying the seed tables between the CPU and GPU memories, we iteratively run the GPU kernel code n_i times. Each time the GPU kernel code is run we generate $n' = n_k\, n_{wi}$ pseudo-random numbers. Thus, the total number of pseudo-random numbers generated is given by $n = n_i\, n'$.

Therefore, in our model we have four separate parts to consider: (1) the time to upload the seed tables, T_{seedUp}; (2) the time to download the seed tables $T_{seedDown}$; (3) the kernel execution time, T_k; and (4) the time to download the n' pseudo-random numbers, $T_{GPU\rightarrow CPU}(n')$. Combining these factors, the time to generate n pseudo-random numbers using the GPU, $T_{RN}^{GPU}(n)$, can be expressed as:

$$\begin{aligned} T_{RN}^{GPU}(n) &= T_{seedUp} + T_{seedDown} + \\ &\quad n_i\,[T_k(n_k,n_{wgs}) + T_{GPU\rightarrow CPU}(n')] \\ &= T_{seedUp} + T_{seedDown} + \\ &\quad n_i\,[T_k(n_k,n_{wgs}) + t_{GC}\,n' + t_s]\,. \end{aligned} \tag{6}$$

The seed table size is 28 words, and we require a separate seed table for each work group. However, given that the number of work groups is at most in the thousands, and the number of pseudo-random numbers that we will be copying back from the GPU is typically in the millions, we can ignore T_{seedUp} and $T_{seedDown}$ in equation (6) and use the following approximation,

$$T_{RN}^{GPU}(n) \approx n_i\,[T_k(n_k,n_{wgs}) + t_{GC}\,n' + t_s]\,. \tag{7}$$

Also note that in our formulation of $T_k(n_k,n_{wgs})$, we tacitly assumed that there are an unlimited number of compute units available on the GPU. In practice, of course, the GPU architecture has limited number of compute units, n_{CU}. When the number of work groups is larger than the number of compute units, the extra work corresponding to these additional work groups must be scheduled sequentially on the GPU. To take this effect into account, we introduce a modified kernel execution time function, $T_k^*(n_k,\, n_{wgs},\, n_{wg})$. The modified kernel execution time can be expressed as

$$T_k^*(n_k,\, n_{wgs},\, n_{wg}) = T_k(n_k,n_{wgs})\,\lceil n_{wg} / n_{CU} \rceil\,. \tag{8}$$

To compute the speedup of the GPU accelerated algorithm, we need a model for the running time of the sequential algorithm on the CPU. As this running time should depend linearly on the number of pseudo-random numbers generated, we model the time to generate the numbers using the CPU, $T_{RN}^{CPU}(n)$, as:

$$T_{RN}^{CPU}(n) = t_c^{CPU}\, n. \tag{9}$$

The pseudo-random number generator used on the GPU is an implementation of RANLUX [8]. Thus, on the CPU we use the GNU Scientific Library (GSL) implementation of this pseudo-random number generator [9]. For the GSL implementation of RANLUX with a luxury level of 0 (gsl_rng_ranlxs0) the value measured for t_c^{CPU} for the Athena system used for our experiments is 50*ns/number*. For a luxury level of 2, this value was measured to be 120*ns/number*. Combining the two models, the speedup of generating n pseudo-random numbers using the GPU compared to CPU can be computed as

$$S_{RN}(n) = T_{RN}^{CPU}(n) \,/\, T_{RN}^{GPU}(n). \tag{10}$$

This speedup is presented below in Figure 4. In this figure we show both the experimentally measured speedup (on the Athena system) and the theoretical speedup based on the model presented above computed using the machine parameters for the Athena system. Note that we plot the theoretical speedup for two models. The first model assumes an unlimited number of compute units, and the second model is based on equation (8), where the number of compute units is limited to 14 (as for the Athena system).

How we obtain the increasing number of work items in Figure 4 requires some additional explanation. We use work group sizes, n_{wgs}, of 32, 64, and 96 and we fix the number of iterations, n_i, to 10. We adjust n_{wg} and n_k to generate the same number of pseudo-random numbers (98,304,000) for different n_{wgs} in the following manner. The number of work groups, n_{wg}, is increased from 2^0 to 2^9. When n_{wgs} is 32, 64 and 96, n_k is respectively varied from 600×2^9 to 600×2^0, from 300×2^9 to 300×2^0, and from 200×2^9 to 200×2^0.

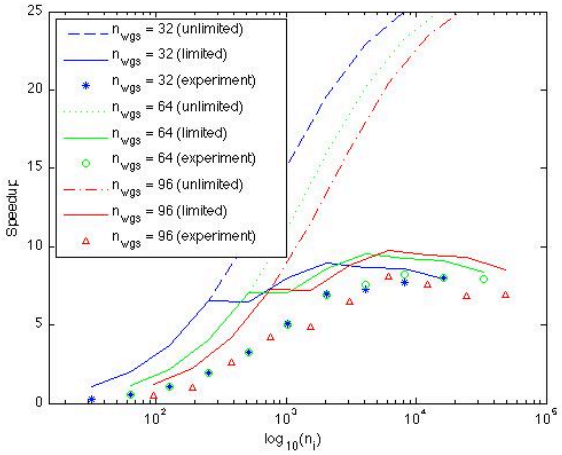


Figure 4. Speedup plots comparing the GPU execution time to the CPU execution time for the pseudo-random number generation library. Three different work group sizes (32, 64 and 96) are used. The speedup results are plotted as a function of the number of work items. As explained in the text, the number of work groups and the number of kernel cycles are both varied in order to compute the same number of pseudo-random numbers for each data point. These results are for RANLUX with luxury level 0.

Note that the experimentally measured results agree well with the modified theoretical model. We include in the plot the theoretical model for an unlimited number of compute units. As one can see, the overall speedup is limited and ultimately approaches an asymptotic value. This limit results primarily from the time required to copy back the pseudo-random numbers from the GPU to the CPU memories.

3.2. PMC Speedup Analysis

We can compute a speedup for the RHT simulation using a hybrid-computing scheme (i.e., using the GPU to compute the pseudo-random numbers) relative to using solely the CPU. First, we consider the time required on a single thread using the CPU to compute the pseudo-random numbers. Let the required number of pseudo-random numbers be denoted by n. Note that the number of pseudo-random numbers required increases linearly with the number of photons used in the RHT simulation, although the constant of proportionality depends on the specific geometry and problem parameters (e.g., the absorption coefficient as a function of position). We can express the sequential simulation time, $T_S(n)$, as

$$T_S(n)=T_{RN}^{CPU}(n)+T_{RHT}(n), \tag{11}$$

where $T_{RHT}(n)$ is a time required for the radiative heat transfer simulation (excluding the pseudo-random number generation) and $T_{RN}^{CPU}(n)$ is the time for generating pseudo-random numbers using the CPU.

An analysis of the time required for the RHT simulation running in parallel using multiple threads and a single GPU to compute the pseudo-random numbers is more complex as it involves the parallel execution of the thread managing the GPU and the GPU program itself [7]. As discussed in the preceding subsection, the time to compute a pseudo-random number on the GPU is a complex function of a number of factors specific to the GPU used and its configuration. We can, however, simplify the analysis by considering only a single parameter, the block size B. As a general rule, the efficiency of computing the numbers increases with the block size. However, the exact efficiency depends on the number of work items, n_{wi}, the number of iterations, n_i, and the number of kernel cycles, n_k (as discussed above and in detail in reference [7]). Given these parameters, the block size B is given by the formula

$$B = n_k\, n_i\, n_{wi}. \tag{12}$$

As noted earlier, the number of work items, n_{wi}, can be configured in a number of different ways by selecting different values for the number of work groups, n_{wg} and the work group size, n_{wgs} as the number of work items is the product of these two parameters. We assume that these parameters are chosen to maximize the efficiency of computation on the GPU. Given these definitions, the parallel simulation time, $T_P(p,pn,B)$, can be expressed as

$$\begin{aligned} T_P(p,np,B) = {} & pT_{RN}^{GPU}(B) + \\ & \max\{p(\lceil n/B\rceil-1)\, T_{RN}^{GPU}(B), \\ & \lfloor n/B\rfloor\, T_{RHT}(B) + T_{RHT}(n-\lfloor n/B\rfloor B)\}, \end{aligned} \tag{13}$$

where $T_{RN}^{GPU}(B)$ is a time to fill in the pseudo-random numbers using the GPU for the block size B, and p is the number of threads.

The timeline for the RHT simulation for a single thread in the above equation is illustrated in Figures 5 and 6. Figure 5 shows the simulation time when the pseudo-random number block generation time takes longer than the RHT time; note how the RHT part of the code is blocked while it waits for a new block of pseudo-random numbers is generated by the GPU.

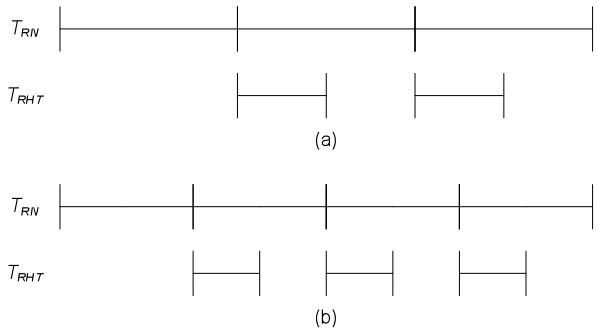


Figure 5. A timeline showing what the RHT thread and the pseudo-random number thread manager are doing relative to each other when $T_{RN}(B)$ is longer than $T_{RHT}(B)$. In (a) we show the case the block size is larger than in (b).

To illustrate where the "max" arises in equation (12), in Figure 6 we illustrate the simulation time when the $T_{RN}(B)$ (the time to generate a block of pseudo-random numbers on the GPU) is shorter than the time $T_{RHT}(B)$.

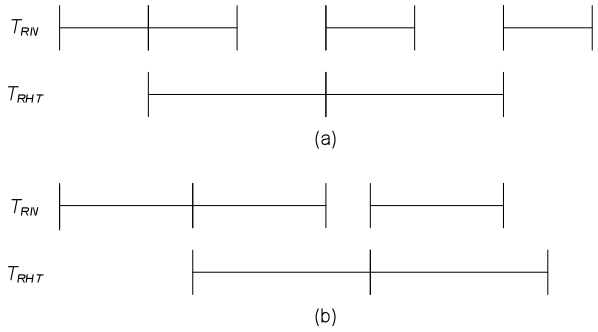


Figure 6. Illustrations of the simulation timeline showing the RHT thread and the pseudo-random number managing thread. In (a) the block size is large enough that $T_{RN}(B)$ can generate a new block before $T_{RHT}(B)$ completes. In (b) we illustrate the case when the block size is even larger than in (a).

In Figure 7 we illustrate the case when the simulation is multi-threaded with multiple threads consuming the pseudo-random numbers generated. Again, note how the "max" in equation (12) is used to describe this case.

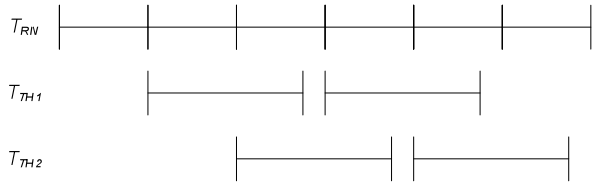


Figure 7. An example illustrating multiple threads executing the RHT part of the code using the pseudo-random numbers generated from the single thread managing the GPU.

Finally, the overall scaled speedup *SS* for the hybrid version the RHT simulation using the GPU-accelerated libraries can be expressed as:

$$SS(p,np,B) = p\ T_S(n) / T_P(p,np,B). \qquad (14)$$

4. Experimental Results

For the experimental results presented in this section, we use the Athena system in Virginia Tech [10]. Athena is a cluster system with GPUs and large RAM memory. The system is made up of 16 nodes and each node consists of 4 octa-cores. Each node also has one NVIDIA S2050 GPU. Each GPU contains 14 compute units, and each compute unit consists of 32 processing elements [11]. For all the results presented in this section, a luxury level of 2 is used for the RANLUX pseudo-random number generators.

To measure the scaled speedup on this hybrid architecture, the CPU-only version of the simulation is first run and timed on a single CPU. These measurements are compared to the running time of the simulation on the hybrid architecture for a scaled instance of the problem. In Figures 8 and 9 we show the speedup of the RHT simulation using the GPU accelerated random number generator and a single CPU as compared to the solely CPU-based version. In Figure 8, note that when the number of photons used by the simulation is small, the scaled speedup is limited by the time required to generate the first block of pseudo-random numbers (e.g., see the timelines in Figures 6). For larger numbers of photons, this initial time is amortized as many blocks are used. Note that by using a smaller block size, this transition to improved speedup occurs for a smaller number of photons. The results in Figure 8 correspond to the timelines shown in Figure 6 where the asymptotic speedup is determined by the relative amount of time spent in RHT portion of the simulation.

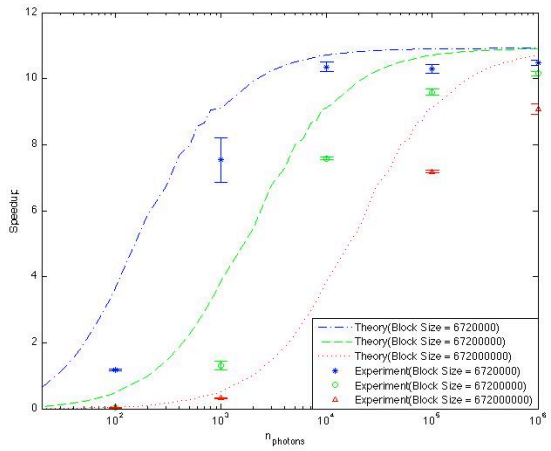


Figure 8. The speedup for the GPU-accelerated simulation run on a single CPU for large block sizes. The photon numbers are increased from 10^2 to 10^6 in order to vary the workload. The work group size is 96, the number of work groups is 14, and the number of kernel cycles is 500. To change the block size, the number of iterations is respectively set to 10, 100 and 1000.

In Figure 9 we show the speedup for the timeline case shown in Figure 5. In this case, the asymptotic speedup is limited by the speedup of pseudo-random numbers being generated on the GPU. As shown in the figure, this speedup decreases with smaller block sizes.

The scaled speedup obtained with multiple threads on a single node is in Figure 10. This plot shows that the overall performance of the simulation increases almost linearly. As with the results in Figure 8, the speedup is limited for small numbers of photons by the generation of the initial blocks of pseudo-random numbers. This fact is illustrated in the timelines shown in Figure 7. Again, as in Figure 8, this initial cost is amortized as multiple blocks are used for larger numbers of photons.

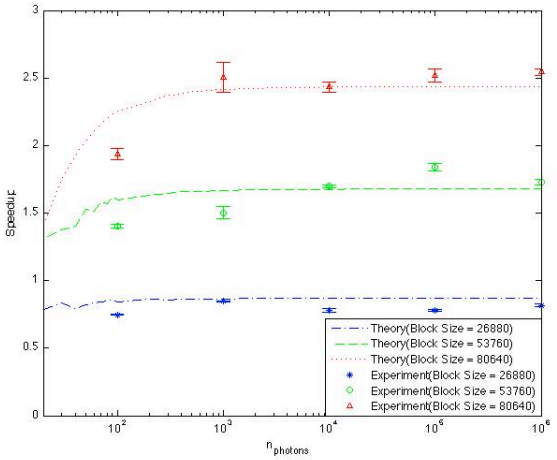


Figure 9. The speedup of the RHT simulation for on a single CPU for small block sizes. The photon numbers are increased from 10^2 to 10^6 in order to vary the workload. The work group size is 96, the number of work groups is 14 and the number of iterations is 1. To change the block size, the number of kernel cycles is set to 20, 40 and 60.

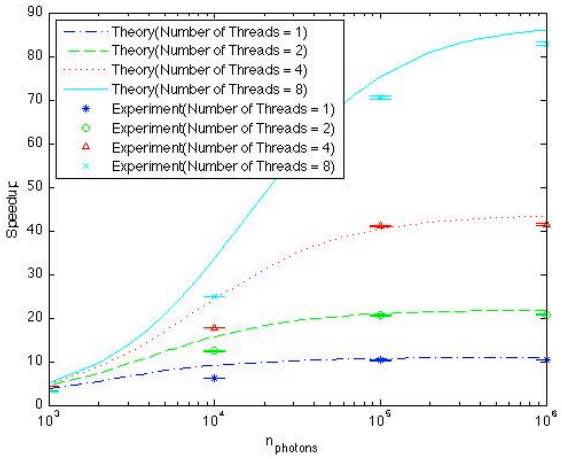


Figure 10. The measured scaled speedup for GPU-accelerated version of the RHT simulation using 1, 2, 4 and 8 threads on a single node (i.e., with one GPU). The photon numbers are increased from 10^3 to 10^6 in order to vary the workload. The work group size is 96, the number of work groups is 14, the number of iterations is 100 and the number of kernel cycles is 500.

The relative cost of the RHT portion of the simulation can be varied by increasing the computed accuracy of the energy contributed to an element during the transmission of a photon (as depicted in Figure 1(c)). This increased accuracy requires the use of more pseudo-random numbers for the Monte Carlo integration involving the absorptivity when traversing the element. The effect of this increased accuracy on the speedup is shown in Figure 11. The speedups for three different numbers of samples per element (80, 160, and 240) are shown in this figure.

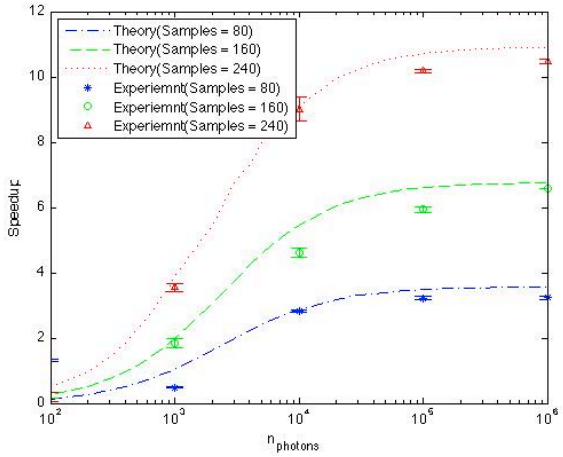


Figure 11. The speedup for the GPU-accelerated algorithm run on a single CPU for different transmission sampling strategies (as described in the text). The number of samples per element is respectively set to 80, 160 and 240. The number of photons used is increased from 10^2 to 10^6 in order to vary the workload. For these results the work group size is 96, the number of work groups is 14, the number of iterations is 100 and the number of kernel cycles is 500.

To be able to generate the theoretical curves shown in Figure 8-11, one needs to know how the RHT portion of the simulation, $T_{RHT}(B)$, scales with block size. In Figure 12 we show how this time varies with block size and with differing numbers of samples per element during the transmission calculation.

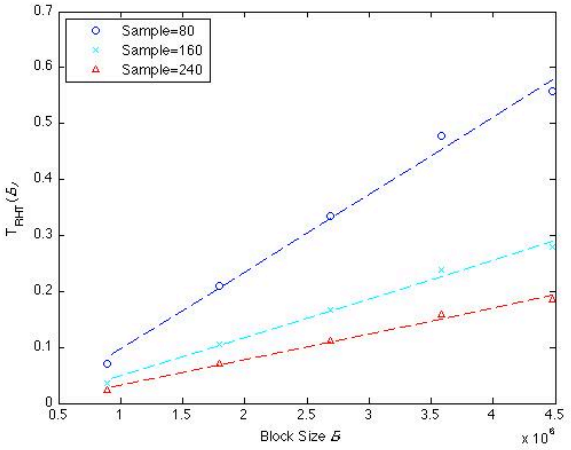


Figure 12. The change of $T_{RHT}(B)$ as a function of block size B. The block size B is increased from 896,000 to 4,480,000 for the data shown on this plot. Linear least squares fits to these data points are shown as the dashed lines in this figure. For sampling of 80, 160 and 240 points per element, the respective slopes from the least squares fit are 1.38×10^{-8}, 6.9×10^{-9} and 4.6×10^{-9}.

Finally, the RHT simulation was run on a complete hybrid architecture including distributed memory (using 10 nodes), using multiple threads on each node (8 threads per node), and using the GPU-accelerated pseudo-random number generator (using one GPU per node). The scaled speedup results obtained on this hybrid architecture are shown in Figure 13. Clearly, significant scaled speedups

can be obtained for a modestly sized hybrid architecture using this approach.

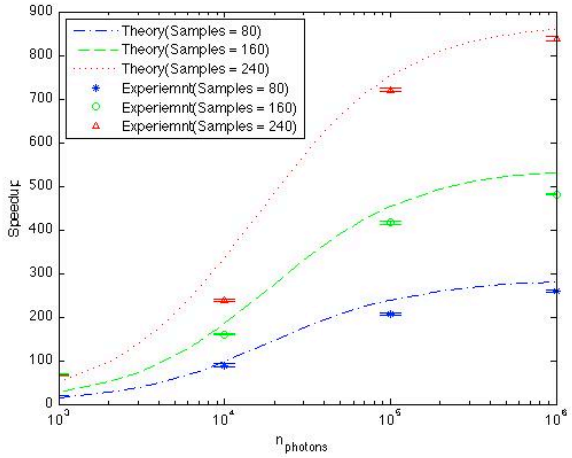


Figure 13. The scaled speedup plots for the RHT simulation using the GPU-accelerated pseudo-random number generator on a hybrid computing architecture. The number of photons used in the Monte Carlo simulation is increased from 10^3 to 10^6 in order to vary the workload. 10 compute nodes are used with 8 threads per node for the simulation. For the GPU the work group size is 96, the number of work groups is 14, the number of iterations is 100 and the number of kernel cycles is 500.

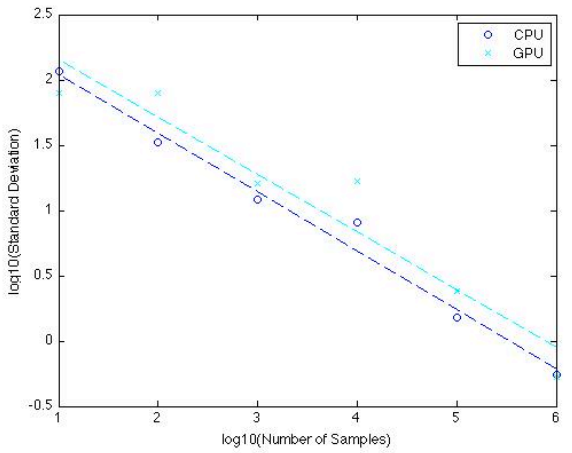


Figure 14. The standard deviation for the mean temperature obtained on different compute nodes as a function of the number of samples. Note that the slope of the best-fit line in this graph is -0.5, which is consistent with the Monte Carlo simulation data on different nodes being statistically independent.

To ensure that the simulation results obtained on this parallel system are statistically independent when using multiple nodes, we computed the mean of the temperature at one point in the interior of the domain for the multiple threads within each node. We then compute the standard deviation of these means (from the 10 nodes). This standard deviation is plotted in Figure 14 as a function of the number of photons used in the simulation. The slope of this graph is -0.5, which is what one would expect (from the central limit theorem) and is consistent with statistically independent simulation results from different nodes. Note that this result does not prove statistical independence, but it does show that the data is not statistical dependent.

5. Conclusions

In this paper, we have implemented a Monte Carlo application framework on a hybrid computer architecture where GPU acceleration is used for generating large blocks of pseudo-random numbers asynchronously. The target application is a baseline Radiative Heat Transfer (RHT) simulation; we presented experimental results that confirm our analysis about the scaled speedup of this approach. Overall, this approach can be very effective and achieve nearly a 1,000 times speedup on a modestly sized hybrid machine.

Our approach demonstrates how GPU acceleration can be used in real world applications such as RHT. We observe that transferring computation from the CPU to GPU can significantly improve the simulation efficiency. However, the overhead of memory copies to and from the CPU and GPU must be amortized through the use of large data block transfers and significant data reuse on the GPU.

6. ACKNOWLEDGEMENTS

This work was supported by NSF grant CCF-0728901.

7. REFERENCES

[1] R. G. dos Santos, *et al.*, "Coupled large eddy simulations of turbulent combustion and radiative heat transfer," *Combustion and Flame,* vol. 152, pp. 387-400, 2008.

[2] R. D. Boudreau, "A solution to the integral equations for radiative transfer of heat in the atmosphere," Ph.D. thesis, Texas A&M University, College Station, 1968.

[3] B. G. Wiedner and C. Camci, "Technique for the determination of local heat flux on steady state heat transfer surfaces with arbitrarily specified external and internal boundaries," 29th National Heat Transfer Conference, pp. 21-31, 1993.

[4] M. F. Modest, Radiative heat transfer, second edition. Academic Press, 2003.

[5] M. K. Drost and J. R. Welty, "Monte Carlo simulation of radiation heat transfer in arrays of fixed discrete surfaces using cell-to-cell photon transport," 28th National Heat Transfer Conference and Exhibition, pp. 85-91, 1992.

[6] I. Veljkovic and P. Plassmann, "Scalable Photon Monte Carlo Algorithms and Software for the Solution of Radiative Heat Transfer Problems," High Performance Computing and Communication (HPCC), vol. 3726, pp. 928-937, 2005.

[7] F. James, "RANLUX: A Fortran implementation of the high-quality pseudo-random number generator of Lüscher", Computer Physics Communications, 79 (1994) 111–114

[8] "GNU Operating System GSL(GNU Scientific Library Library[online].http://www.gnu.org/software/gsl/.Accessed:Aug.1.2011

[9] J. H. Lee, Mark T. Jones, Paul E. Plassmann, "A Hybrid software framework for the GPU acceleration of multi-threaded Monte Carlo Applications," International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), vol. 1, pp. 70-76, 2011

[10] "Athena"[online].http://www.arc.vt.edu/arc/Athena/index.php. Accessed: Dec.1.2011

[11] "TESLA™ S2050 GPU Computing System"[online]. http://www.nvidia.com/docs/IO/105880/NV-DS-Tesla-S2050-Aug11.pdf. Accessed: Dec. 1. 2011.

Classical Mechanical Hard-Core Particles Simulated in a Rigid Enclosure using Multi-GPU Systems

D.P.Playne and K.A. Hawick
Computer Science, Institute for Information and Mathematical Sciences,
Massey University, North Shore 102-904, Auckland, New Zealand
email: { d.p.playne, k.a.hawick }@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

March 2012

ABSTRACT
Hard-core interacting particle methods are of increasing importance for simulations and game applications as well as a tool supporting animations. We develop a high accuracy numerical integration technique for managing hard-core colliding particles of various physical properties such as differing interaction species and hard-core radii using multiple Graphical Processing Unit (m-GPU) computing techniques. We report on the performance trade-offs between communications and computations for various model parameters and for a range of individual GPU models and multiple-GPU combinations. We explore uses of the GPU Direct communications mechanisms between multiple GPUs accelerating the same CPU host and show that m-GPU multi-level parallelism is a powerful approach for complex N-Body simulations that will deploy well on commodity systems.

KEY WORDS
m-GPU; GPUDirect; N-Body; hard-core collisions; polydisperse radii; multi-species.

1 Introduction

Models based upon classical N-Body particle systems [2] are commonly used at various levels of approximation in game simulations [5,11,12] but still play an important role in understanding physical phenomena such as diffusion, phase mixing and separation [4], and other behaviours that arise from specific geometric distributions.

Considerable work has been reported in the literature on uses of molecular dynamics whereby approximate potential models are used to simulate atomic and molecular systems [14, 30, 36]. A recent review by Larsson and co-workers [26] points out that there is still considerable scope for improved algorithms and for hybrid solutions to the molecular dynamics N-Body applications problem.

Another N-Body application area of significance is in simulating astrophysical phenomena including: general relativity systems [44]; cosmological simulations [37]; simulations related to dark matter theories [25]; and other modifications to Newtonian gravity [39]. A body of recent important work is progressing in this field where simulation provides an important means of exploring the implications of various theories of dark matter.

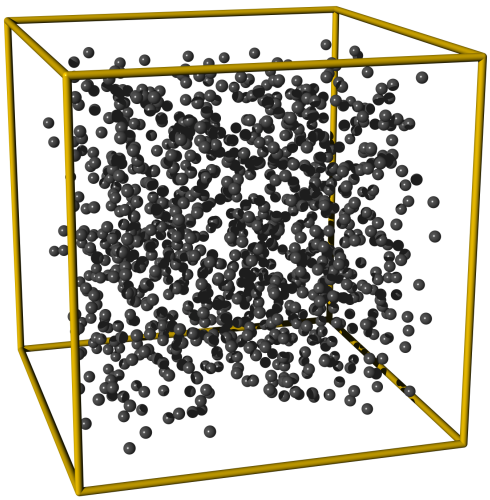


Figure 1: Three-dimensional particle system with rigid walls.

It is therefore of continuing importance to understand the computational performance for N-Body particle system simulations and indeed such simulations have found use as benchmark kernels for high-performance computing systems [8, 19, 20, 23, 29, 33, 38].

In the limit of a large number of particles and a thermodynamically equilibrated system a simple benchmark rating such as number of particle updates per second suffices, but in practice, many gaming situations in which particle models are employed the system is quite definitely not in equilibrium. Much present research into understanding growth and transient behaviour in physical, chemical and biolog-

ical systems is also concerned with models and systems that are far from equilibrium.

It is therefore interesting to add to the parametric space of a particle integration benchmark by considering how the load balance and computational organisation can be changed by varying a manageable parameter such as the temperature of a simulated system.

We consider a system of hard spherical particles that interact at long and medium range through an attractive potential with a soft repulsive core, but with a hard impenetrable core at very short range. This is a realistic model for many scenarios and allows us to experiment with a range of computational intensities in the resulting particle benchmark. The system is also usefully realistic – it can be modelled with a definite walled container and not with the unrealistic periodic boundaries often used in physics model simulations used for studying statistical mechanical properties.

Two typical paradigms for deploying N-Body simulation codes [34] are as either near teal-time interactive codes that have a built in visualisation capability, or as batch-oriented non-interactive codes, that may offer a frame dumping capability but which are not intended nor capable of rendering in real time.

There are a number of sophisticated data organisational methods that approximate the $\mathcal{O}(N^2)$ interactions with appropriate potential cut-offs which can be managed as spatial trees and neighbour lists [3, 41]. Some of these approaches can be optimised for particular architectures such as hypercubic systems [6, 15]. Collision dynamics can also be optimised using appropriate collision lists and associated techniques [10].

The size of systems that are feasible to simulate obviously depends on the desired simulated time period as well as most critically – upon the number of particles involved. At the time of writing, work has been reported on up to $N \approx 10^10$ possible particles in a gravitational simulation involving collisional and collisionless systems [7]. Articles such as that by Aarseth report work with more complex models such as post Newtonian corrections [1] with $N \approx 10^5$ particles simulated on on GRAPE [22] type computer architectures.

Various architectural approaches have been applied to this important class of application problems, including clusters and volunteer distributed computing systems [9] and conventional multi-core CPU systems modern multicore CPU systems [40]. Work by Groen and co-workers [16] indicates the typical number of cores (60-750) that have been recently feasibly employed in multicore cluster systems to tackle a single N-Body application simulation cold dark matter.

In this present paper we are interested in the capabilities of graphical processing units and systems deploying single and multiple GPU systems and several authors and research groups have reported work on N-Body calculations on GPUs – although nearly always on a very specific algorithm [14, 17, 21, 42, 43].

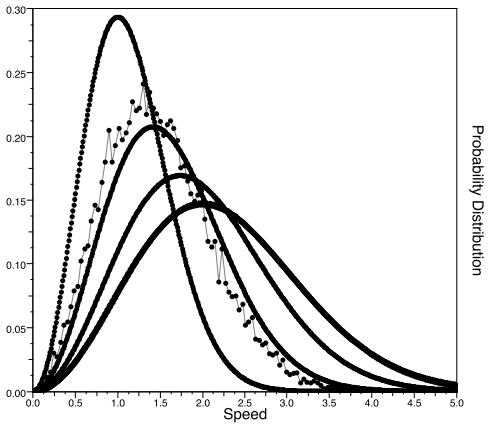


Figure 2: Theoretical Maxwell-Boltzmann distribution computed for Temperatures 1,2,3,4 with $k_B = m \equiv 1$ from equation 2 showing increasing most-probable speed and distribution width with increasing temperature. Experimental data from a sample simulation is included for reference.

A considerable range of sophisticated physics quality simulation codes for specific and special purpose N-Body applications are available and have been compared in the review by Fortin and co-workers [13].

We are interested in the computational structure of a hybrid N-Body application that involves different sorts of short or long range potentials, hard core collisions and also interactions with enclosing rigid walls of a container. This covers the principal algorithms needed for both games quality physics simulations as well as more sophisticated energy conserving statistical mechanics studies. We are also interested in being able to deploy very accurate high-order numerical integration methods such as the Hairer 10th order time stepping algorithm [18].

In this paper we explore how a sophisticated N-Body simulation can be developed for multiple GPU systems with the option of choosing collisional or collisionless systems, and parameterising the model using temperature T as well as the number of individual particles N.

2 Distribution of Speeds

It becomes feasible and realistic to ascribe a temperature to an ensemble of simulated particles once the system size N is large enough. In practice a few thousand particles yields thermal measurements that can be compared with the theoretical definition discussed below.

The scalar speed is defined in terms of the velocity com-

ponents for each particle as:

$$v = \sqrt{(v_x^2 + v_y^2 + v_z^2)} \quad (1)$$

and taking the Boltzmann probability energy factor e^{-E/k_BT} alongwith kinetic energy expression for a point particle $\frac{1}{2}mv^2$ we obtain the probability density distribution of speeds as:

$$f(v) = 4\pi \left(\frac{m}{2\pi k_B T}\right)^{3/2} v^2 \exp(-mv^2/k_B T) \quad (2)$$

which is normalised so that:

$$\int_0^\infty f(v)dv \equiv 1 \quad (3)$$

and is known as the Maxwell-Boltzmann distribution with the form as shown in Figure 2. Differentiating, we locate the most probably speed at the peak of the distribution:

$$\frac{df(v_p)}{dv} = 0 => v_p = \sqrt{(2k_B T/m)} \quad (4)$$

where v_P is the most probable occurring speed and thus temperature T can be obtained from:

$$T = \frac{mv_p^2}{2k_B} \quad (5)$$

Temperature is also related to the width of the Maxwell-Boltzmann distribution functions and can also be (more reliably) fitted to that, than from the peak position which is more prone to experimental uncertainty.

3 Molecular Dynamics

The molecular dynamics (MD) technique involves a numerical integration of the classical equations of motion for a number of particles, or molecules, given a model Hamiltonian [2, 24, 27]. The resulting numerical trajectory through phase space, can be used to make a number of useful measurements of the dynamical properties of a system [31].

There are several difficulties involved in applying this technique to the simulation of an alloy, even a purely binary one. Primarily, the first requirement is for a suitable model Hamiltonian for the system. While this is not easy to obtain exactly, it *is* possible to construct an approximate one with the basic properties, using pair-wise potentials. For example one very simple model would be to assume a system of soft spherical atoms in a box, with two atomic species present, and atoms of each species preferentially attracting atoms of its own species.

A model Hamiltonian of the Lennard-Jones form [27] can be constructed using:

$$U(R) = 4\epsilon \left[\left(\frac{\sigma}{R}\right)^{12} - x^{i,j}\left(\frac{\sigma}{R}\right)^{6}\right] \quad (6)$$

This gives the potential energy U due to the pair-wise interaction of two atoms of species i, j, separated by some radial distance $R = |\mathbf{r}_i - \mathbf{r}_j|$, given two atom specific parameters in the form of an energy ϵ and a length scale σ, and a cross term coupling fraction $x^{i,j}$ which is greater for interactions between like species i, j than for opposite species.

4 Model Implementation

This molecular dynamics simulation has been implemented for multi-GPU systems using CUDA. The simulation computes the potential between particles using the all-pairs algorithm and computes hard-sphere collisions using a posteriori method. All hard-sphere collisions are inelastic as are the collisions of particles hitting the walls of an enclosing box. A diagram of the inelastic collisions are shown in Figure 3.

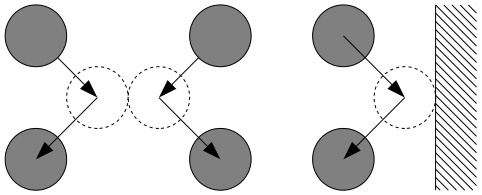


Figure 3: Inelastic Collisions

This simulation has been implemented and tested for m-GPU systems containing up to four GPU devices. The high-level algorithms are the same as the ones discussed in [19]. The particles are evenly distributed between the GPU devices and each device is responsible for updating its particles. To calculate the total force on a particle, each device will calculate the force each of it's own particles exert on each other as well as the force exerted on its particles by the particles stored in the other devices. This requires communication between the different devices, there are a number of ways this communication can be performed and they are discussed and compared in section 5. All force calculations make use of the tiling algorithm [29] as it still provides the best performance.

Various methods can be used to integrate the motion of the particles based on the laws of motion and the potential between them. The different methods have various tradeoffs in terms of memory usage, computation time, stability and accuracy. Selecting the optimal method is not always simple. It has been shown that high-order methods may be computationally more expensive per step, they are stable and accurate with larger step sizes resulting in an overall improvement in performance. The implementations in this work have been tested using the following methods - Euler, Runge-Kutta 2^{nd}, Runge-Kutta 4^{th}, Dormand-Prince 5^{th} and Hairer 10^{th}.

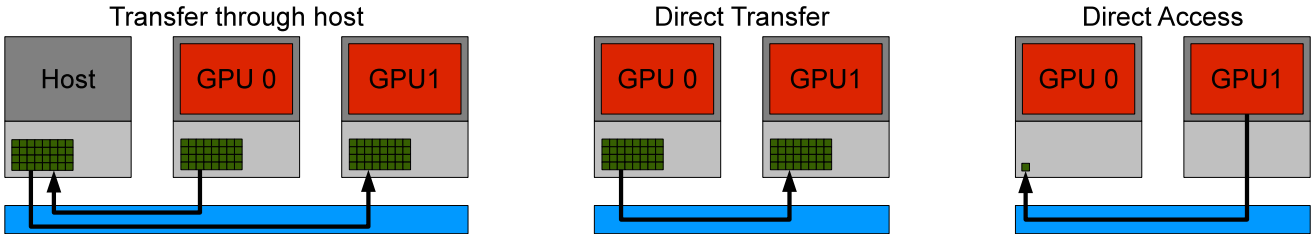


Figure 4: GPU transfer methods.

The fundamental collision model used in this research is the same as the method discussed in [19]. After each time-step, each particle is checked against every other particle to determine if a collision has occurred. The collision detection phase is similar to the all-pairs force calculation. If a collision has occurred, the device will calculate the time at which the collision occurred and record it. Once all the particles have been checked, the collision that occurred first will be corrected. Both particles are stepped backwards in time to the point of collision, an inelastic collision is computed at this point and the particles are then moved forward in time to same time as the rest of the system. This process is repeated until there are no more collisions.

5 M-GPU Communication

The focus of this research was to compare the different methods of communication CUDA provides for transferring data between devices. In [19] we evaluated the performance of PCIe extender chassis such as the Dell C410x. For that comparison the simple peer-to-peer memory copy made available by NVIDIA's GPUDirect 2.0. In this research we compare the performance of all the peer-to-peer communication methods.

Before GPUDirect, all communication in an m-GPU system had to go through the host memory. GPUDirect 2.0 allows the direct transfer of data between devices and eliminates the need for extra system memory and transfer overhead. There are two types of direct communication supported by GPUDirect 2.0 - Direct Transfer and Direct Access.

Direct Transfer is a host controlled memcpy from one device directly to another. This data transfer does not need to go through the host and thus avoids unnecessary transfer overheads. CUDA 4.0 provides two methods for Direct Transfer - `cudaMemcpyPeer` and `cudaMemcpyPeerAsync`. `cudaMemcpyPeer` is a blocking memory call that will not return until the transfer is complete. `cudaMemcpyPeerAsync` is non-blocking and the method will return immediately even though the transfer has not yet completed.

Direct Access allows a thread executing on one device to access a value stored in the memory of another device. This feature makes used of the Unified Virtual Addressing (UVA) supported by CUDA 4.0. UVA gives a single address space to the host and device memory, previously the host and each device had a separate memory address space [28]. Direct Access is designed for Non-Uniform Memory Access (NUMA) patterns. Using this method of communication the kernels on each device can simply access the values from the other devices and there is no need for any memcpy calls.

6 CUDA Implementations

To use the CUDA peer-to-peer communication methods, the devices must have Peer Access enabled. This must be set on each device for every other device peer-to-peer communication will be used. Peer-to-peer communication with device `d` can be enabled by simply calling the function `cudaDeviceEnablePeerAccess(d,0)`.

Listings 1 and 2 show the two methods of Direct Transfer. Listing 1 blocking copy method `cudaMemcpyPeer`. The thread connects two each device and uses `cudaMemcpyPeer` to copy the particle data out of another device into memory on the current device. Once all devices have performed this copy, kernels will be launched on each device to compute something with this data. This computation can either be a force calculation or a collision detection kernel.

Listing 1: Direct Transfer Method - cudaMemcpyPeer.

```
for(int t = 1; t < P; t++) {
  for(int id = 0; id < P; id++) {
    int id2 = (id + t)%P;
    cudaSetDevice(id);
    cudaMemcpyPeer(p2[id],id,p[id2], id2,size);
  }
  for(int id = 0; id < P; id++) {
    cudaSetDevice(id);
    //Compute something using p2[id]
  }
}
```

The non-blocking copy method is very similar but will use

`cudaMemcpyPeerAsync` instead. In this implementation the host will launch all the memory copy calls at once instead of waiting for each one to finish before launching the next. The kernels are also launched immediately but will not execute until the data transfer is completed. The code snippet for this data transfer method is shown in Listing 2.

Listing 2: Direct Transfer Method - cudaMemcpyPeerAsync.

```
for(int t = 1; t < P; t++) {
  for(int id = 0; id < P; id++) {
    int id2 = (id + t)%P;
    cudaSetDevice(id);
    cudaMemcpyPeerAsync(p2[id],id,p[id2], id2,
                        size, stream[id]);
  }
  for(int id = 0; id < P; id++) {
    cudaSetDevice(id);
    //Compute something using p2[id]
  }
}
```

The Direct Access method shown in Listing 3 does not use any copy functions. Instead the computation kernels can simply access the particles in the other devices. Each device simply calculates the device id of all the other devices and get the address of their particle data. The computation kernels can then access the particles at these addresses. This requires Unified Virtual Addressing otherwise each device would access its own memory and not the memory on other devices.

Listing 3: Direct Access Method.

```
for(int id = 0; id < P; id++) {
  cudaSetDevice(id);
  id1 = (id + 1)%P;
  id2 = (id + 2)%P;
  ...
  //Compute something using p[id1], p[id2]...
}
```

These three implementations can all be used to transfer data between multiple devices for molecular dynamics simulations on m-GPU systems. The performance of these three methods are compared in Section 7.

7 Performance Results

The implementations discussed in this work have been evaluated using two different experiments. The first shows the time required to compute a molecular dynamics simulation with potentials and hard-sphere collisions at different system densities. The second compares the performance of the different m-GPU implementations. Both of these experiments have been computed with four NVIDIA C2070 compute cards connected to a Dell C410x PCIe extender chassis and use the Runge-Kutta 4^{th} order integration method. The performance of this system is compared to other GeForce and Tesla card configurations in [19].

The first experiment was to compare the three peer-to-peer transfer methods - Synchronous Direct Transfer (DT-S), Asynchronous Direct Transfer (DT-A) and Direct Access (DA). These three methods have been tested on the Dell C410x and the results for a range of transfer sizes are shown in Figure 5. This shows that the Asynchronous Direct Transfer method provides the best performance and Direct Access outperforms Synchronous Direct Transfer for small transfers but is slightly slower for larger transfers ($>$ 256 KB). It is important to note that the Direct Transfer methods communicate the information between devices but it must still be read by the kernels from global memory, whereas the kernels using Direct Access will have already read the required values.

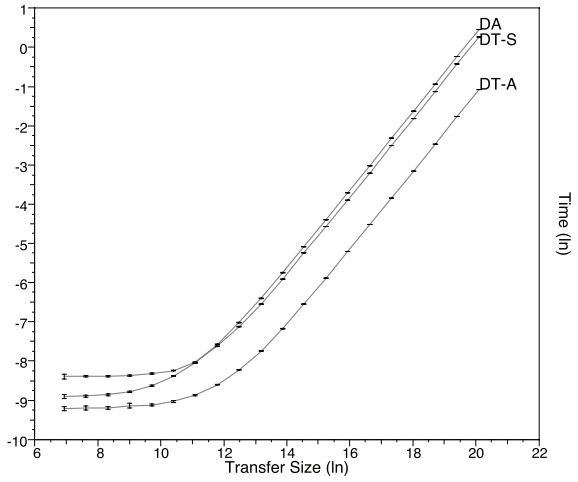


Figure 5: Transfer time for three peer-to-peer communication methods for 4KB-2GB shown in ln-ln scale.

The second experiment is designed to compare the performance of the three m-GPU implementations in the actual simulation. The methods have been compared across a range of system sizes and densities. The performance results (in milliseconds per time step) are presented for one fixed density. These results are shown in Table 1.

8 Discussion

The almost indistinguishable performance results of the three implementations presented in Section 7 are surprising to say the least. The close performance of the two Direct Transfer methods is not entirely unusual as the data is transferred as a block in by both implementations. In previous work [35], with asynchronous data transfer through host showed a significant performance improvement over

Table 1: Performance comparision of GPUDirect simulation implementations. Times acurate to ± 0.005 millisecs per timestep.)

Size N	Direct Transfer	Direct Transfer (Async)	Direct Access
1024	3.46	2.66	3.52
2048	5.55	4.76	5.58
4096	9.81	9.03	9.80
8192	18.78	18.12	18.73
16384	53.38	52.26	53.18
32768	194.56	193.50	193.78
65536	729.27	727.75	727.01

synchronous. However, this work focused on lattice models where only the borders had to be transferred and thus all communication could be completely hidden.

However, it was highly unexpected that the Direct Access method would provide almost the same performance as the other two implementations. It was expected that the NUMA style of access to another device's memory would significantly reduce performance. However, all performance benchmarks and tests for both all-pairs force calculation and collision detection show almost no difference in performance. The high performance of Direct Access represents a significant step forward for m-GPU programming. Previous m-GPU research [32,35] which used communication through the host found the performance of m-GPU programs to be extremely sensitive to communication methods used. Getting the best performance out of such applications required a great deal of fine-tuning.

For this reason, the high-performance of the Direct Access functionality of GPUDirect 2.0 was unexpected. However, it has performed well in every test and significantly reduces programmer effort. Kernels can simply access data from other devices without the need for explicit data transfer and extra memory to store duplicated data. For this reason we believe the Direct Access functionality of GPUDirect 2.0 to be extremely valuable for m-GPU systems.

9 Conclusions

The molecular dynamics simulation discussed in this work is capable of collisional and collision less simulations, with or without elastic wall collisions and a range of possible force/potential models including simple gravitational attraction and multiple species Lennard Jones style systems. These simulations have been implemented for m-GPU systems with multiple GPUs to accelerate a single CPU. Such systems are becoming an increasingly important node architecture for future supercomputers and clusters.

These implementations make use of the GPUDirect 2.0 device-device communication methods that allow data to be directly transferred from one device to another without going through the host. The three implementations show almost no difference in performance despite the fact that the Direct Access method uses a Non-Uniform Memory Access pattern. This surprising result makes it significantly easier to develop efficient m-GPU applications and represent a major step forward in m-GPU technology.

References

[1] Aarseth, S.J.: Post-newtonian n-body simulations. Mon. Not. R. Astron. Soc. 378, 285–292 (2007)

[2] Allen, M., Tildesley, D.: Computer simulation of liquids. Clarendon Press (1987)

[3] Barnes, J., Hut, P.: A hierarchical o(n log n) force-calculation algorithm. Nature 324(4), 446–449 (1986)

[4] Biferale, L., Coveney, P.V., Ubertini, S., Succi, S.: Discrete simulation of fluid dynamics: Applications. Phil. Trans. Roy. Soc. A 329, 2384–2386 (2011)

[5] Bourg, D.M.: Physics for Game Developers. No. ISBN 978-0596000066, O'Reilly (2002)

[6] Brunet, J.P., Mesirov, J.P., Edelman, A.: An optimal hypercube direct n-body solver on the connection machine. In: Proc. Supercomputing 90. pp. 748–752. IEEE Computer Society, 10662 Los Vaqueros Circle, CA 90720-1264 (nov 1990)

[7] Dehnen, W., Read, J.I.: N-body simulations of gravitational dynamics. arXiv 1105.1082v1, University of Leicester (May 2011)

[8] Demiroz, B., Topcuoglu, H.R., Kandemir, M., Tosun, O.: Particle simulation on the cell be architecture. Cluster Comput. July, 1–14 (2011)

[9] Desell, T., Magdon-Ismail, M., Szymanski, B., Varela, C.A., Willett, B.A., Arsenault, M., Newberg, H.: Evolutionary n-body simulations to determine the origin and structure of the milky way galaxy's halo using volunteer computing. In: Proc. IEEE Int. Symp. on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW). pp. 1888–1895 (16-20 May 2011)

[10] Donev, A., Torquato, S., Stillinger, F.H.: Neighbor list collision-driven molecular dynamics simulation for non-spherical hard particles: I. algorithmic details. J. Comput. Phys. 202(2), 737–764 (2005)

[11] Eberly, D.H.: 3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics. No. ISBN: 978-0122290633, Morgan Kaufmann (2006)

[12] Eberly, D.H., Shoemake, K.: Game Physics. No. ISBN 978-1558607408, Morgan Kaufmann (2003)

[13] Fortin, P., Athanassoula, E., Lambert, J.C.: Comparisons of different codes for galactic n-body simulations. Astronomy and Astrophysics 531, A120–1–11 (2011)

[14] Ganesan, N., Bauer, B.A., Lucas, T.R., Patel, S., Taufer, M.: Structural, dynamic, and electrostatic properties of fully hydrated dmpc bilayers from molecular dynamics simulations accelerated with graphical processing units (gpus). J. Computational Chemistry July, 2958–2973 (2011)

[15] G.Fox, M.Johnson, G.Lyzenga, S.Otto, J.Salmon,

D.Walker: Solving problems on concurrent processors, vol. 1. Prentice Hall (1988)

[16] Groen, D., Zwart, S.P., Ishiyama, T., Makino, J.: High performance gravitational n-body simulations on a planet-wide distributed supercomputer. arXiv 1101.0605v1, Leiden Observatory, Leiden University, The Netherlands (2011)

[17] Hagan, R., Cao, Y.: Multi-gpu load balancing for in-situ visualization. In: Int. Conf. on Parallel and Distributed Processing Techniques and Applications (2011)

[18] Hairer, E.: A Runge-Kutta Method of Order 10. J. Inst. Maths. Applics. 21, 47–59 (1978)

[19] Hawick, K.A., Playne, D.P.: Hard-sphere collision simulations with multiple gpus, pcie extension buses and gpu-gpu communications. Tech. Rep. CSTN-138, Computer Science, Massey University, Albany, Auckland, New Zealand (August 2011)

[20] Hawick, K., Playne, D., Johnson, M.: Numerical precision and benchmarking very-high-order integration of particle dynamics on gpu accelerators. In: Proc. International Conference on Computer Design (CDES'11). No. CDE4469, Las Vegas, USA (July 2011)

[21] Hu, Q., Syal, M., Gumerov, N.A., Duraiswami, R., Leishman, J.G.: Toward improved aeromechanics simulations using recent advancements in scientific computing. In: Proc. 67th Annual Forum of the American Helicopter Society. Virginia Beach, USA (3-5 May 2011)

[22] Hut, P., Makino, J.: Astrophysics on the GRAPE Family of Special-Purpose Computers. Science 283, 501–505 (1999)

[23] Kavanagh, G.D., Lewis, M.C., Massingill, B.L.: GPGPU planetary simulations with CUDA. In: Proceedings of the 2008 International Conference on Scientific Computing (2008)

[24] Kittel, C.: Introduction to Solid State Physics. Wiley (2004), iSBN 978-0-471-41526-8

[25] Koda, J., Shapiro, P.R.: Gravothermal collapse of isolated self-interacting dark matter haloes: N-body simulation versus the fuid model. Mon. Not. R. Astron. Soc. March, 1–14 (2011)

[26] Larsson, P., Hess, B., Lindahl, E.: Algorithm improvements for molecular dynamics simulations. Wiley Interdisciplinary Reviews: Computational Molecular Science 1, 93–108 (January 2011)

[27] Lennard-Jones, J.: Cohesion. Proc. Royal Soc. 43, 461–482 (1931)

[28] NVIDIA® Corporation: CUDA™ 4.0 Programming Guide (2011), http://www.nvidia.com/, last accessed November 2011

[29] Nyland, L., Harris, M., Prins, J.: Fast n-body simulation with cuda. In: Nguyen, H. (ed.) GPU Gems 3, chap. 31. Addison Wesley Professional (August 2007)

[30] Pawley, G.S.: Molecular dynamics simulation of the plastic phase; a model for SF6. Molecular Physics 43(6), 1321–1330 (1981)

[31] Pawley, G.S.: Molecuar dynamics and spectroscopy. In: Price, D.L., Skold, K. (eds.) Methods in experimental physics. pp. 441–519. Academic Press (1986), volume 23 Neutron Scattering Part A

[32] Playne, D.P., Hawick, K.A.: Comparison of GPU Architectures for Asynchronous Communication with Finite-Differencing Applications. Tech. Rep. CSTN-111, Massey University (2010), submitted to: Concurrency and Computation: Practice and Experience

[33] Playne, D.P., Johnson, M.G.B., Hawick, K.A.: Benchmarking GPU Devices with N-Body Simulations. In: Proc. 2009 International Conference on Computer Design (CDES 09) July, Las Vegas, USA. No. CSTN-077 (July 2009)

[34] Playne, D.P.: Notes on Particle Simulation and Visualisation. Hons. thesis, Computer Science, Massey University (June 2008)

[35] Playne, D., Hawick, K.: Hierarchical and Multi-level Schemes for Finite Difference Methods on GPUs. In: Proc. CCGrid 2010, Melbourne, Australia. No. CSTN-099 (May 2010)

[36] Sarman, S., Evans, D.J.: Heat flow and mass diffusion in binary lennard-jones mixtures. Phys.Rev.A 45(4), 2370–2379 (feb 1992)

[37] Schneider, M.D., Cole, S., Frenk, C.S., Szapudi, I.: Fast generation of ensembles of cosmological n-body simulations via mode-resampling. arXiv 1103.2767v3, Lawrence Livermore Nat. Lab., USA. (May 2011), lLNL-JRNL-471523

[38] Stock, M.J., Gharakhani, A.: Toward efficient GPU-accelerated N-body simulations. In: in 46th AIAA Aerospace Sciences Meeting and Exhibit, AIAA 2008-608 (January 2008)

[39] Suzuki, T.: Method of N-body simulation on the MOdified Gravity. arXiv 1110.5420, Yamaguchi University, Japan (October 2011)

[40] Tanikawa, A., Yoshikawa, K., Okamoto, T., Nitadori, K.: N-body simulation for self-gravitating collisional systems with a new SIMD instruction set extension to the x86 architecture, Advanced Vector eXtensions. New Astronomy 17, 82–92 (2012)

[41] Warren, M.S., Salmon, J.K.: A parallel hashed oct-tree n-body algorithm. In: Supercomputing. pp. 12–21 (1993), citeseer.ist.psu.edu/warren93parallel.html

[42] Yokota, R., Barba, L.A.: Fast multipole method vs. spectral methods for the simulation of isotropic turbulence on gpus. arXiv 1110.2921v1, Boston University, USA (October 2011)

[43] Yokota, R., Barba, L.A.: Fast n-body simulations on gpus. arXiv 1108.5815, Boston University, USA (August 2011)

[44] Zhao, G.B., Li, B., Koyama, K.: N-body Simulations for f(R) Gravity using a Self-adaptive Particle-Mesh Code. Phys. Rev. D 83, 044007–1–19 (2011)

In-Situ Data Compression for Flow Simulation in Porous Media

Henry Lehmann[1], **Bernhard Jung**[1]
[1]Virtual Reality and Multimedia Group, TU Bergakademie Freiberg, Freiberg (Saxony), Germany

Abstract— *Supercomputing simulations easily generate datasets in the range of tera or even peta bytes. However, writing datasets during simulation and reading them for post-processing implies a high I/O load and thus imposes a major bottleneck. In the computation phase, compute nodes stall at writing large amounts of data and the simulation slows down - scientists have to capture data less frequently losing important information. In the post-processing phase, the data has to be read into memory and visualization becomes a time consuming batch process. These problems can be addressed with in-situ data reduction techniques which reduce the I/O load significantly. Recently a novel in-situ data compression method, ISABELA, has been proposed [6]. It provides high compression rates at negligible run time overhead while being local, communication-free, and scalable. This contribution describes a variant of the ISABELA method and its application to a fluid simulation in porous media. Modifications to the original algorithm concern the specific properties of porous media as well as a general new method for exploiting temporal coherence in time-dependent fluid simulation data. Evaluation results concern the optimal choice of curve fitting parameters for data compression and give insight into the trade-off between desired error bounds and compression rates. Overall, the very high compression rate of the ISABELA method is confirmed by our experiments.*

Keywords: in-situ processing, lossy compression, high performance computing, b-spline, data-intensive application, temporal pattern

1. Introduction

As high performance computing resources and algorithms become generally available to scientists, the simulation of physical phenomena plays an increasingly important role in prototyping and knowledge discovery. However the growing computational power raises new problems and challenges to existing hard- and software [2]. With numerical simulations producing amounts of data in the order of tera or even peta bytes, storage devices are driven to peak performance on file systems not optimized for parallel operation [14]. Similarly, for later post-processing, large amounts of data have to be read into memory. This makes post-processing a time consuming batch process which hardly can be influenced by the user once being started [8].

In-situ data reduction is a promising approach to deal with the heavy I/O load of high-resolution supercomputing simulations. In in-situ processing, data compression techniques are directly integrated into the computation phase of the simulation run. Instead of forcing scientists to write out only a subset of the computed simulation data, in-situ processing offers the potential of storing simulation data in their full resolution [8], [6].

The work in this paper is specifically motivated by the need for in-situ data reduction in large-scale Lattice Boltzmann simulations [13] of flow in porous media. Porous media are characterized by highly complex geometries, which present special challenges to numerical simulations. In the Collaborative Research Center (CRC) 920 [1] at Technical University Bergakademie Freiberg (Germany), multifunctional filters for metal melt filtration are researched. The aim is to contribute towards the development of zero-defect materials improving safety of road and railway vehicles as well as aircrafts and other applications requiring highly stressable metal based components.

A further goal is to provide a basis for the design of a new real-time visualization framework using immersive Virtual Reality, replacing offline batch processing by interactive data exploration experiences. With interactive data visualization playing an increasingly important role in data analysis, in-situ data reduction is seen a key feature in CRC 920's evolving simulation and visualization frameworks.

Recently, a novel in-situ data reduction technique called ISABELA (for "In-situ Sort-And-B-spline Error-bounded Lossy Abatement") has been proposed [6]. It outperforms existing lossy and lossless compression techniques for scientific data and offers surprisingly high compression rates while introducing only a minor overhead in computation time into the simulation.

This paper reports on a variant of the ISABELA method (as no implementation details are given in [6]) and its application to fluid simulation data of flow within porous media. In Section 2, we review related work on lossless and lossy compression of scientific data. Section 3 presents our variant of the ISABELA compression method. Our ISABELA-variant takes advantage of differentiating between geometry and fluid cells when applied to voxel grids of porous media. We further describe an error quantization scheme and investigate a novel general method for exploiting temporal coherence in fluid simulation data to enhance com-

pression ratio. In Section 4, we present compression results and analyze different parameterizations of the compression algorithm. Finally, we conclude that ISABELA-like methods provide very good compression results for flow simulation in porous media as well. However, the optimal parameterization and implementation details for our melt simulation dataset varies from the suggested parameterization in the original ISABELA description.

2. Related Work

The complex internal structure of scientific data makes compression a challenge for existing compression methods. Lossless techniques for fast online compression of floating point data are presented e.g. in [10] and [3]. These techniques function by predicting data values and storing small differences between predicted and original data values. They also offer options to omit precision bits supporting lossy compression. However, at high compression rates the relative error increases significantly [6]. Moreover, the compression is not block-based making them unsuitable for random read/write applications.

In the context of compression of medical images, [11] introduces a lossless method based on wavelets improved by detection of global and local symmetries. However, this method induces significant computational effort to enable high compression rates and is thus not suited for in-situ data reduction.

Generally, scientific floating point data is commonly known to be inherently noisy and random-like, high-entropy data without repeating bit or byte patterns, thus commonly considered to be hardly compressible [15]. As argued in [6], standard lossless compression methods do not reach satisfying compression rates or otherwise introduce significant computational overhead.

Due to approximations and discretizations, simulation data is already flawed. Thus, lossy compression with a user-defined error bound is a reasonable choice for achieving high compression rates.

A lossy data reduction technique for "In-situ Sort-And-B-spline Error-bounded Lossy Abatement" (ISABELA) was recently proposed [6]. It outperforms existing lossy compression techniques, like Wavelets [5], and lossless compression techniques, like FPC [3], LZMA and Bzip. ISABELA offers surprisingly high compression rates of up to $\sim$85% while introducing only a minor overhead in computation time and ensuring a user-defined error bound. It is designed for local encoding and decoding of scientific data.

The main idea of ISABELA is to apply a pre-conditioner, which sets up a very strong signal-to-noise ratio by sorting data values. Due to monotonic and smooth behavior of the sorted data, a regression model approximates the data with significantly fewer coefficients compared to the overall amount of data [6], [4].

Relatively few studies have been conducted on the application of B-splines for compression of scientific data. In [4] and [7] B-splines are used to fit scattered data by optimizing positions of control points. ISABELA extends this concept and transforms the data to guarantee an accurate fitting model.

3. Compression Algorithm

In this section, a variant of the ISABELA-algorithm for in-situ data reduction is presented. As in the original proposal, a first pre-processing step is to subdivide, linearize and sort the data; in our variant, the algorithm further accounts for special properties of porous media, by differentiating between geometry and fluid cells. Then, the sorted data is fitted through cubic B-splines. A new variable width error quantization scheme is applied, which bounds the maximum relative error on a per-point basis. Finally a new method for exploiting the temporal coherence of fluid simulation data is integrated into the compression algorithm.

3.1 Subdividing, Linearizing and Sorting

As first pre-processing step, the data is subdivided into smaller contiguous blocks that can be compressed and decompressed independently from each other, thus supporting random access of data blocks. By linearizing the data, the three-dimensional compression problem is transformed into a one-dimensional one. Sorting of the linearized data results in smooth curves, that can be approximated much better through B-splines [6].

In figure 1 the procedure of subdividing, linearizing and sorting the data is sketched. In step 1 the entire data grid is subdivided into smaller blocks, which contain N data values. In step 2 the blocks are linearized in a fixed serialization order. This gives a scattered sequence $\alpha = (a_1, \ldots, a_N)$ of N data values for each block, which is shown in step 3. In step 4 the data is sorted in ascending order to get a monotonic sequence $\alpha' = (a'_1, \ldots, a'_N)$. In general, the method is independent of a particular block size N.

In simulations of porous media, the data grid contains geometry cells, which are represented by a constant data value $\bar{a}$ in α'. This offers a further opportunity for compression. Discarding data values belonging to geometry cells in the sorted sequence α' yields a new sequence of $n \leq N$ data values, which is denoted as $\beta = (b_1, \ldots, b_n)$. This enhances the performance of the compression, but keeps the possibility to reconstruct the data on per-block basis without decompressing the entire dataset. The sequence β then is applied to a regression model, which is able to represent the data accurately.

To be able to revert the sorting process and reconstruct α during decompression, one has to keep track of the indices, when sorting and shortening α to obtain β. The shortened sequence of indices of elements of β in α is denoted as $\tau = (t_1, \ldots, t_n)$. Remaining values with indices not being

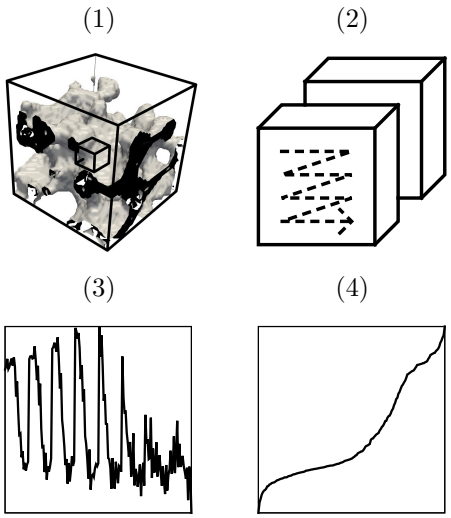


Fig. 1: Illustration of pre-processing the data for compression: *(1)* subdividing voxel grid containing porous media into smaller blocks, *(2) + (3)* linearizing data in blocks into scattered sequence, *(4)* sorting scattered sequence yields monotonic smooth data.

in τ, are set to the constant $\bar{a}$ in order to reconstruct the geometry cells omitted in β. In following an example of a particular instantiation of α, α', β and τ, with geometry cells having $\bar{a} = 0$ is given:

$$\begin{array}{lcccccccc} \alpha & : & 0.3 & 0.1 & 0.0 & 0.2 & 0.4 & 0.5 & 0.3 & 0.0 \\ \alpha' & : & 0.0 & 0.0 & 0.1 & 0.2 & 0.3 & 0.3 & 0.4 & 0.5 \\ \beta & : & 0.1 & 0.2 & 0.3 & 0.3 & 0.4 & 0.5 & & \\ \tau & : & 2 & 4 & 1 & 7 & 5 & 6 & & \end{array} \quad (1)$$

3.2 Cubic B-Spline Fit

The sequence β has a high signal-to-noise ratio and can be fitted by a cubic B-spline with few control points accurately [6].

Originally developed in the CAD and computer graphics fields, B-splines provide a flexible framework for representing complex shapes [9]. Cubic B-splines are constructed from a set of control points and a knot vector. The control points are used to specify the shape of the resulting curve, whereas the knot vector defines the influence of the control points on piecewise polynomial curve segments.

Cubic B-splines are denoted as a sum of M control points p_j, weighted by B-spline basis functions N_j^3 of degree 3:

$$S(u) = \sum_{j=1}^{M} N_j^3(u) \cdot p_j \quad (2)$$

The knot vector is chosen to have equally spaced knots, which means each control point has the same influence on the curve. B-splines are able to locally adapt the shape of a monotonic point cloud accurately, which makes them a very flexible and efficient tool in the present case of data compression [6], [4].

In order to fit the sequence β using the above setup, the n data values b_i are mapped to uniformly distributed values $x_i \in [0, 1]$. For each point a linear equation $S(x_i) = b_i$ according to eqn. (2) is set up, which gives a linear system

$$\mathcal{N}_n \cdot \mathcal{P} = \beta \quad (3)$$

$\mathcal{N}_n$ holds B-spline basis functions $N_j^3(x_i)$ for n equations and $\mathcal{P}$ holds control points p_j. The system is solved using the least squares method. This involves computing the Moore-Penrose Pseudoinverse $\mathcal{N}_n^+$. The control points, which minimize least square distances are given by $\mathcal{P} = \mathcal{N}_n^+ \cdot \beta$.

In the case of $n \leq M$, the observed block has a high amount of geometry cells. Then the system is underdetermined and the data will be stored directly, since the number of data values is less or equal to the number of control points.

Since the number n of data values per block is varying, the number of equations in the linear system 3 also varies. To address this problem, the matrices $\mathcal{N}_n$ and $\mathcal{N}_n^+$ are precomputed for $n = M+1, \ldots, N$ and the corresponding case is selected at compressing and decompressing respectively.

Once the data has been fitted, eqn. 2 is used to reconstruct the data from control points $\mathcal{P}$ and x_i values, which yields an approximation of β. The approximate sequence is denoted as $\beta' = (b'_1, \ldots, b'_n)$ and is obtained by $\beta' = \mathcal{N}_n \cdot \mathcal{P}$. In the next step the approximation error $b_i - b'_i$ is handled by applying error correction.

3.3 Error Quantization

Cubic B-spline regression ensures an accurate fitting model for the data. However, the approximation β' does not meet the precision needed for scientific data analysis. Thus an (unspecified) additional error quantization step is applied in ISABELA [6]. In the following, we describe the error quantization method developed for our ISABELA variant.

A user-defined error criterion is met by setting an upper bound p for the relative error ϵ_i on a per-point basis. The relative error is measured between sequence β and $\gamma = (c_1, \ldots, c_n)$:

$$\epsilon_i = \frac{(b_i - c_i)^2}{b_i^2 + \bar{\epsilon}} \leq p \quad (4)$$

γ denotes the sequence of data, reconstructed from the approximate sequence β' by applying an error correction step. Since the relative error can grow without limit for values near zero, the maximum relative error is bounded by introducing $\bar{\epsilon}$ into eqn. 4 [12]. The error correction step is given by the following quantization scheme

$$c_i = b'_i + \delta_i \cdot \bar{c} \cdot \Delta(b'_i) \quad (5)$$

where δ_i denotes integer quantization steps and $\bar{c} \cdot \Delta(b'_i)$ denotes the real valued width of the quantized value grid.

The quantization scheme in eqn. 5 uses a variable quantization width depending on b'_i, since a global width guaranteeing $\epsilon_i \leq p$ can become very small having δ_i to grow very high.

$\Delta(b'_i)$ describes the maximum tolerance, so that the relative error between b'_i and $b'_i \pm \Delta(b'_i)$ is less or equal to p. The variable quantization width follows from eqn. 4 and is given by

$$\Delta(b'_i) = p \cdot \sqrt{b'^2_i + \bar{\epsilon}} \tag{6}$$

Since original data values b_i from β are not known at decompressing, B-spline approximations b'_i near b_i are used to predict the quantization width. Since $\Delta(b'_i)$ does not bound the relative error with respect to b_i, a scaling factor $\bar{c}$ is applied to ensure, that

$$\bar{c} \cdot \Delta(b'_i) \leq \Delta(b_i) \tag{7}$$

holds, which guarantees eqn. 4 to hold for c_i computed from eqn. 5. The scaling factor is obtained by

$$\bar{c} = \min_i \frac{\Delta(b_i)}{\Delta(b'_i)} \tag{8}$$

Given β, β' and $\bar{c}$, the integer quantization steps are computed by

$$\delta_i = \left\lfloor \frac{b_i - b'_i}{\bar{c} \cdot \Delta(b'_i)} + \frac{1}{2} \right\rfloor \tag{9}$$

To store δ_i efficiently a stream of B_δ bit numbers is used. The number of bits needed to store δ_i is determined by

$$B_\delta = \left\lceil \mathrm{ld}\left(\max_i \delta_i\right) \right\rceil \tag{10}$$

Using τ the values c_i of γ are reordered into $\gamma' = (c'_1, \ldots, c'_N)$, which denotes an approximation of α meeting bounded relative error criterion. The reconstruction is accomplished for $i = 1, \ldots, n$ by setting $c'_{t_i} = c_i$. Remaining values with indices i not being in τ are set to $c'_i = \bar{a}$ to reconstruct geometry cells.

3.4 Exploiting Temporal Coherence

In ISABELA, temporal data patterns are exploited to enhance compression ratio. The improvement results from similarities in the index sequence τ, which changes only slightly over consecutive timesteps on a per block-basis. The pattern is applied by introducing a reference sequence τ^0 and delta sequences $\tau^1, \tau^2, \ldots$. Instead of storing the indices of delta sequences directly, the differences $\tau^1 - \tau^0, \tau^2 - \tau^0, \ldots$ are stored. This yields mainly small integer numbers, which are compressed using a lossless compression technique. Using this procedure the compression performance of ISABELA is improved by $\sim 3\%$.

In our implementation, we exploit another temporal pattern, which relies on the continuous nature of the data in time dimension. Since linearized sequences α of consecutive timesteps change smoothly, difference encoding of data values can be applied. For this, α^0 is considered a reference sequence and following sequences $\alpha^1, \alpha^2, \ldots$ are considered delta sequences. Temporal smoothness is sketched in figure 2.

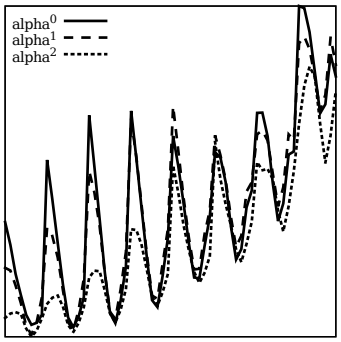


Fig. 2: Illustration of temporal smoothness of consecutive sequences α^0, α^1 and α^2 of u data for one block.

Instead of compressing the data values of delta sequences directly, differences of data values to decompressed reference sequence $\bar{\alpha}^1 = \alpha^1 - \gamma'^0, \bar{\alpha}^2 = \alpha^2 - \gamma'^0, \ldots$ are taken into account. The resulting values $\bar{\alpha}^k = (\bar{a}^k_1, \ldots, \bar{a}^k_N)$ of a delta sequence with index $k > 0$ are small. Applying sorting, linearizing and B-spline regression yields shortened sequences $\bar{\beta}$, $\bar{\beta}'^k$ and τ^k belonging to $\bar{\alpha}^k$.

Now, the relative error between $\bar{b}^k_i$ and $\bar{c}^k_i$ has to be bounded by p on a per-point basis. The error quantization is accomplished by

$$\begin{array}{rcl} c^k_i & = & c'^0_{t^k_i} + \bar{b}'^k_i \quad + \quad \delta_i \cdot \bar{c} \cdot \Delta(c'^0_{t^k_i} + \bar{b}'^k_i) \\ \underbrace{\bar{c}^k_i}_{c^k_i - c'^0_{t^k_i}} & = & \bar{b}'^k_i \quad + \quad \delta_i \cdot \bar{c} \cdot \Delta(c'^0_{t^k_i} + \bar{b}'^k_i) \end{array} \tag{11}$$

to meet the criterion. As $\Delta(c'^0_{t^k_i} + \bar{b}'^k_i) \approx \Delta(b'^k_i)$ and $\bar{b}^k_i - \bar{b}'^k_i < b^k_i - b'^k_i$, the integer quantization steps

$$\delta_i = \left\lfloor \frac{\bar{b}^k_i - \bar{b}'^k_i}{\bar{c} \cdot \Delta(c'^0_{t^k_i} + \bar{b}'^k_i)} + \frac{1}{2} \right\rfloor \tag{12}$$

become smaller. Thus, B_δ also decreases which reduces the amount of memory needed to reconstruct delta timestep k.

4. Results

The algorithm described in 3 has three parameters: block size N, number of B-spline control points M, and user-defined error threshold p. The parameters influence the compression ratio CR. Assuming 64 bit double precision floating point values, the compression ratio of one block is given by

$$\begin{array}{rcl} CR & = & 1 - \quad [64 \cdot (M+1) + \lceil \mathrm{ld}(N) \rceil \cdot \\ & & \quad (n+2) + B_\delta \cdot n] \cdot {}^1\!/_{(64 \cdot N)} \end{array} \tag{13}$$

Compression ratio is calculated by taking into account $M \cdot 64$ bits for B-spline control points, 64 bits for the scaling factor $\bar{c}$, $\lceil \mathrm{ld}(N) \rceil \cdot (n+2)$ bits for storing τ as well as n and B_δ and $B_\delta \cdot n$ bits for storing error quantization steps.

In order to estimate performance parameters, the algorithm is applied to subgrids of 26 timesteps of simulation data extracted from a Lattice Boltzmann simulation of melt in porous media. The data contains velocity (u, v, w) and density (ρ) values in a $64 \times 64 \times 64$ grid. The entire dataset has a size of $212,992KB$, where each feature has a total size of $53,248KB$.

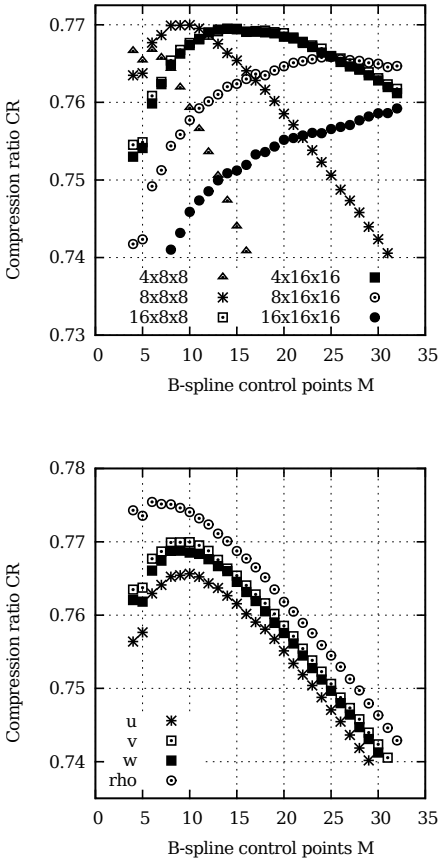


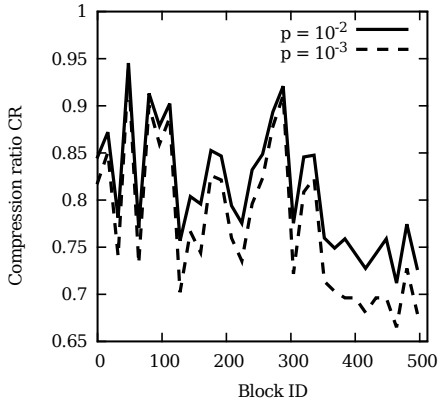
Fig. 3: *Top:* Mean compression rate achieved at compressing u plotted against number of B-spline control points for different block sizes. The optimal compression rate of about 77% was achieved with block size $N = 8 \times 8 \times 8$. *Bottom:* Performance of compressing u, v, w and ρ using block size $N = 8 \times 8 \times 8$. Best compression results over all attributes were achieved for $M = 10$.

4.1 Parameter Analysis

In the compression algorithm as described, the accuracy of cubic B-spline regression primarily depends on the parameters N and M. To resolve the impact of the parameters on the compression ratio, the algorithm was run with varying block sizes and varying number of control points. In figure 3 the results are depicted. We found that $N = 512$ and $M = 10$ yield the best overall compression ratio in our case. We also found that the use of non-cubic block shapes did not improve the compression ratio and we decided to use cubic $8 \times 8 \times 8$ blocks.

On one hand, $N = 512$ is a power of two, which allows to use all $\lceil \mathrm{ld}(N) \rceil$ bits to store the indices in τ optimally. On the other hand, there exists a trade-off between B_δ bits needed to encode the error and the number of control points M. If the data is fitted more accurately by the B-spline, less storage is needed to encode the error quantization steps.

In figure 4, the compression ratio for $N = 512$ and $M = 10$ is given for different user-defined relative error bounds $p = 10^{-2}, 10^{-3}, 10^{-4}$. In parentheses CR for setting $N := n$ is given, which corresponds to not considering geometry cells in the compression rate. In the whole data set 26.9% of the cells are geometry cells. Our experiments show that the algorithm yields high compression ratio ensuring the user-defined error bound. In particular, the algorithm clearly outperforms lossless methods like *gzip* (50.6%), *bzip2* (52.7%) and *lzma* (57.5%), ISABELA-variant algorithms achieve higher compression rates while being local,



p	u	v	w	ρ
10^{-2}	80.2 (71.2)	80.6 (71.7)	80.5 (71.6)	80.9 (70.1)
10^{-3}	76.5 (66.2)	76.9 (66.7)	76.8 (66.5)	77.4 (65.2)
10^{-4}	72.7 (61.0)	73.1 (61.5)	73.0 (61.3)	73.6 (60.1)

Fig. 4: *Top:* Variation of CR per block, implied by changing number of geometry cells and changing number of bits B_δ needed to encode error quantization steps. *Bottom:* Mean compression ratio CR achieved at compressing u, v, w and ρ using $M = 10$ and $N = 8 \times 8 \times 8$ (using variable amount n of data values per block) for different user-defined relative error bounds p. Numbers in parentheses indicate compression rates for non-geometry cells only.

scalable, communication-free and in-situ applicable [6].

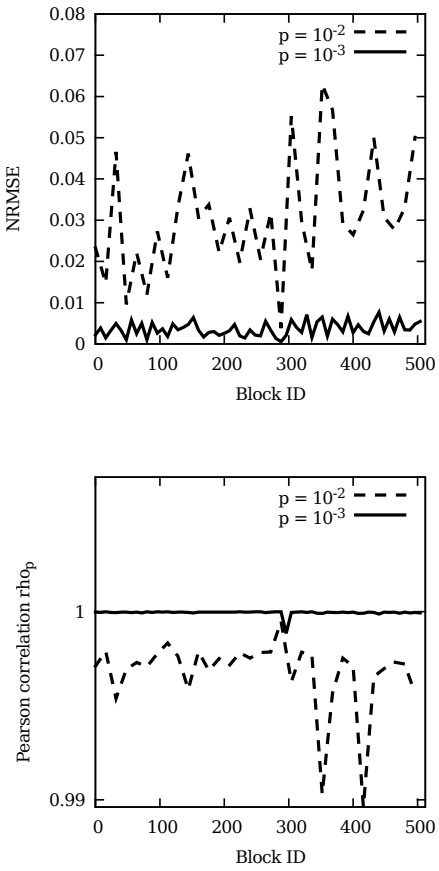


Fig. 5: Performance of the compression algorithm on per-block basis for user-defined relative error bound $p = 10^{-2}, 10^{-3}$. *Top:* normalized root mean square error $NRMSE(\alpha, \gamma')$. *Bottom:* Pearson Correlation $\rho_p(\alpha, \gamma')$.

4.2 Error Analysis

As in [6], normalized root mean square error $NRMSE(\alpha, \gamma')$ and Pearson correlation $\rho_p(\alpha, \gamma')$ are taken into account to observe the performance of the compression algorithm. $NRMSE(\alpha, \gamma')$ and $\rho_p(\alpha, \gamma')$ are given on per-block basis by

$$NRMSE(\alpha, \gamma') = \frac{\sum_{i=1}^{N} \sqrt{(a_i - c'_i)^2}}{\max_j a_j - \min_j a_j} \quad (14)$$

$$\rho(\alpha, \gamma') = \frac{\mathrm{Cov}(\alpha, \gamma')}{\sqrt{\mathrm{Var}(\alpha) \cdot \mathrm{Var}(\gamma')}} \quad (15)$$

Error quantization and regression using the least squares method excellently reconstruct the data and yield high compression ratios even for low error threshold $p = 10^{-2}$. In figure 5, the performance is shown in terms of $NRMSE$ and Pearson Correlation ρ_p. $NRMSE$ shows low values, whereas ρ_p shows values being nearly one, which indicates excellent data reconstruction.

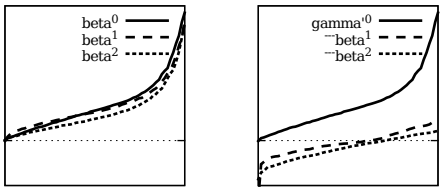


Fig. 6: Impact of difference encoding on the sorted shortened data to be compressed. *Left:* sequences β^0, β^1 and β^2 without difference encoding. *Right:* decompressed reference sequence γ'^0 and difference encoded delta sequences $\bar{\beta}^1, \bar{\beta}^2$. As can be seen, delta sequences are flatter and therefore need less error correction bits.

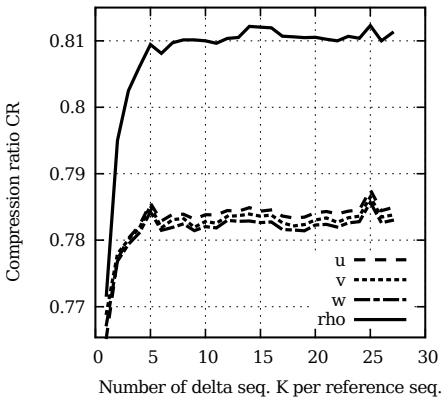
4.3 Analysis of Time Difference Encoding

The pre-conditioning step exploits the continuous nature of the data in spatial dimensions and produces smooth monotonic curves. Applying cubic B-spline regression with few control points accurately fits the data allowing high compression rates. By also taking advantage of the continuous nature of the flow data in temporal dimension, it is possible to further enhance compression ratio. Considering differences between reference sequences and delta sequences allows to reduce the amount of storage needed to encode error quantization steps. The impact of difference encoding data sequences is depicted in figure 6. For consecutive timesteps' sequences, differences are small and therefore, when being compressed with the same relative error bound as the original data, the amount of error correction data is reduced.

The performance of difference encoding K delta sequences $\alpha_1, \ldots, \alpha_K$ per reference sequence α_0 is shown at compressing 26 timesteps of simulation data. The mean compression ratio and the reduction of the number of bits B_δ needed to encode error quantization steps is shown in figure 7. Compression gain ranges from $\sim 2\%$ to $\sim 3\%$ which compares to the gain of the ISABELA method proposed in [6] exploiting temporal pattern in index sequences.

5. Conclusion

We presented a variant of the ISABELA method for in-situ compression of scientific data. One of our modifications to ISABELA addresses specific properties of flow in porous media. Another modification concerns the exploitation of temporal coherence of time-continuous datasets by compressing differences between reference and delta timesteps. For continuous smooth data, compressing differences yields a significant reduction of error correction data. As a (minor) drawback, difference encoding requires decompressing



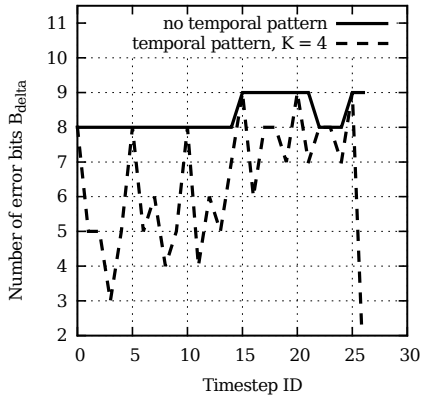


Fig. 7: *Top:* Compressing difference data in temporal dimension enhances mean compression ratio (CR). Plot shows the mean compression ratio achieved at compressing 26 timesteps with varying number of delta sequences K per reference sequence. *Bottom:* Showing reduction of number of bits B_δ needed to encode error quantization steps for one block of u data using difference encoding of $K = 4$ delta sequences.

blocks of reference timesteps. However, the underlying data structures involve B-splines which can be evaluated locally. Thus, local decompression is still possible without significant overhead. Improvements of the overall compression ratio compare to the delta-encoding in the original ISABELA method. Through the combination of both methods, the overall compression rate may be improved even more.

Our results on a melt simulation dataset confirm that high compression rates can be achieved with ISABELA-like algorithms also in the case of porous media. Further, compression lossiness can be controlled comfortably by providing the algorithm with a user-defined error bound. Results were presented that relate several error bound choices to achieved compression rates.

The analysis of our ISABELA variant applied to flow in porous media suggests an optimal block size of $8 \times 8 \times 8 = 512$ data cells. Further, our results indicate that the optimal number of B-spline control points is 10. These results differ from the original ISABELA publication [6] where a block size of 1024 and 30 control points are suggested as optimal. Future work is needed to clarify the optimal parametrizations of ISABELA-like compression methods on different types of datasets.

Acknowledgements

This research was supported by the Deutsche Forschungsgemeinschaft (DFG).

References

[1] Collaborative Research Center 920. Multi-functional filters for metal melt filtration - a contribution towards zero defect materials. TU Bergakademie Freiberg, Freiberg (Saxony), Germany. http://tu-freiberg.de/ze/sfb920/index.en.html.

[2] James Ahrens, Kristi Brislawn, Ken Martin, Berk Geveci, C. Charles Law, and Michael Papka. Large-scale data visualization using parallel data streaming. *IEEE Comput. Graph. Appl.*, 21(4):34–41, July 2001.

[3] Martin Burtscher and Paruj Ratanaworabhan. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Comput.*, 58(1):18–31, January 2009.

[4] Jin J. Chou and Les A. Piegl. Data reduction using cubic rational b-splines. *IEEE Comput. Graph. Appl.*, 12(3):60–68, May 1992.

[5] Jill R. Goldschneider. *Lossy Compression of Scientific Data via Wavelets and Vector Quantization*. PhD thesis, University of Washington, 1997.

[6] Sriram Lakshminarasimhan, Neil Shah, Stéphane Ethier, Scott Klasky, Robert Latham, Robert B. Ross, and Nagiza F. Samatova. Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par (1)*, volume 6852 of *Lecture Notes in Computer Science*, pages 366–379. Springer, 2011.

[7] Seungyong Lee, George Wolberg, and Sung Yong Shin. Scattered data interpolation with multilevel b-splines. *IEEE Transactions on Visualization and Computer Graphics*, 3:228–244, 1997.

[8] Kwan-Liu Ma, Chaoli Wang, Hongfeng Yu, and Anna Tikhonova. In-situ processing and visualization for ultrascale simulations. *Journal of Physics: Conference Series*, 78(1):012043+, July 2007.

[9] Les Piegl and Wayne Tiller. *The NURBS book*. Springer-Verlag, London, UK, UK, 1995.

[10] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. Fast lossless compression of scientific floating-point data. In *Proceedings of the Data Compression Conference*, DCC '06, pages 133–142, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Victor Sanchez, Rafeef Abugharbieh, and Panos Nasiopoulos. 3d scalable lossless compression of medical images based on global and local symmetries. In *Proceedings of the 16th IEEE international conference on Image processing*, ICIP'09, pages 2497–2500, Piscataway, NJ, USA, 2009. IEEE Press.

[12] John Z. Sun and Vivek K. Goyal. Scalar quantization for relative error. *Data Compression Conference*, 0:293–302, 2011.

[13] A. J. Wagner. A practical introduction to the lattice boltzmann method. Department of Physics, North Dakota State University, 2008.

[14] Feng Wang, Qin Xin, Bo Hong, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Tyce T. Mclarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 139–152, 2004.

[15] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.

Symmetry and Simplicity in Simulation: Reducing Complexity in Alternate Parallel–Serial Processing

Clarence Lehman[1] **and Adrienne Keen**[2]
[1]University of Minnesota, 123 Snyder Hall, 1474 Gortner Avenue, Saint Paul, MN 55108, USA
[2]London School of Hygiene and Tropical Medicine, Keppel Street, London WC1E 7HT, UK

Abstract— *Certain simulations are characterized by alternating periods of "expansion" and "contraction." For example, simulated populations of migratory birds may congregate in a central geographic location for overwintering, handled by a single processor, then fan out to dispersed locations for local ecological interactions during the rest of the year, handled by one processor per location. In another application, the difficult problem of fitting parameters to large-scale stochastic simulation models may fan out to numerous processors computing independent stochastic trajectories from the same initial conditions, then "contract" to allow a new, more likely set of parameters to be estimated from the computed distribution of independent trajectories. The contraction is commonly handled by a designated "master processor." In this paper, we point out a simpler, completely symmetric algorithm in which all processors act identically and no processor is designated master. We have used it for applications in simulated annealing and exhibit it here in a standard MPI (Message Passing Interface) environment.*

Keywords: parallel processing, symmetric multiprocessing, parallel–serial simulation, parameter fitting, individual-based modeling

1. Introduction

The goal of this paper is to demonstrate a symmetric technique for coordinating multiple processors and contrast that technique with a more usual master–subordinate technique. We consider the case where multiple processors calculate results independently of one another for an extended length of time, from seconds to minutes or more. The processors then pool their results before partitioning the calculations and expanding to multiple independent processors again. This repeats through multiple expansion–contraction phases until the computation converges to some result. We exhibit the algorithms in detail and illustrate them within an application of parameter fitting.

2. Algorithms

We assume each processor accepts a data structure as input and returns the results of its calculations in the same or another data structure. For simplicity here, we represent this data structure as an array of double-precision floating-point numbers, $local[j]$, though it could take any form. In addition, an array of these data structures, $global[i][j]$, has one row per processor. In the symmetric algorithm, all processors use this array, but in the master–subordinate algorithm, only the master uses it. Any processor can be the master, but here we make it the one numbered 0. The processor number is placed in an integer variable named *cproc* by Function 1 of the appendix. Algorithms in the appendix encapsulate the MPI environment [1] and provide a degree of system independence.

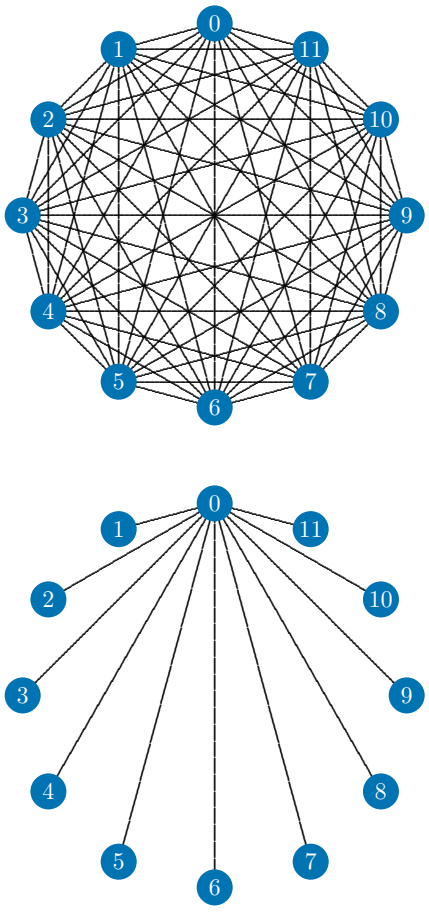


Figure 1. *Communications among a dozen processors in the symmetric technique, top, and the master–subordinate technique, bottom. The standard technique on the bottom might seem simpler, because of fewer interconnections, but asymmetries actually make it more complex.*

We present the algorithms in a stylized form of the language C, as an alternative to pseudocode, so that they define the interface precisely, and so they can be compiled, run, and modified. In the displayed algorithms, flow control and reserved words are bolded, variables and function names are italicized, and certain operations such as '<=', '>=', '!=', and '==' are displayed in a mathematical form as '$\leq$', '$\geq$', '$\neq$', and '$\equiv$', respectively.

2.1 Symmetric technique

In applications of parameter fitting, computing a new set of parameters is a global step, needing input from all the processors together. Therefore, it seems natural to assign that step to a master processor. However, such assignment is unnecessary. If all processors share information equally, then every processor can compute the new parameters for itself, using the same algorithm that would be used by the master processor. No time is lost, for all subordinate processors would be waiting for the master processor anyway. No chance of error arises, for all processors are executing the same code. And a notable simplification results, cutting the number of lines of code needed almost four-fold (see discussion below).

For the symmetric case, the program begins by invoking function *MPBegin* and ends by invoking *MPEnd*, defined in the appendix (steps 1 and 4, respectively, in the algorithm below). Variables t and *tmax* are integers recording the current and the maximum times, respectively. The main loop has five lines.

[1] *MPBegin*();

[2] **for** ($t = 0$; $t \leq tmax$; $t{+}{+}$)
{ *converge* = *NewParameters*(*global*);
if (*converge*) **break**;

[3] *Simulate*(*local*);
MPCommon(*global*, *local*, *W*); }

[4] *MPEnd*();

Steps 1 and 4 above merely begin and end multiprocessing operations. Step 2 loops through the procedure, with each processor invoking function *NewParameters* to handle data from all the processors (array *global*), for example computing a new set of parameters by whatever technique is desired—gradient descent, simulated annealing, genetic search, and so forth. That function returns an indication of whether the process has converged, and if convergence is detected, the program terminates the main loop.

Step 3 invokes function *Simulate*, which carries out the current processor's simulation task, getting its parameters from array *local* and returning its results in the same array. Finally each processor communicates its results to all other processors and symmetrically receives all results back by invoking function *MPCommon* before repeating. W in the example is the width of arrays $global$ and $local$.

In addition to the code for *MPBegin* and *MPEnd* (Functions 1 and 2 in the appendix), this process uses only the five lines of code of Function 3 in the appendix. The amount of data communicated is $WN(N-1)$ elements, where W is the number of elements per processor and N is the number of processors. Processors pass messages only once per iteration.

2.2 Master–subordinate technique

When the global process is assigned to a master processor, the program also begins by invoking function *MPBegin* and ends by invoking *MPEnd*, defined in the appendix (steps 1 and 8, respectively, in the algorithm below). The variable *cproc* defines the processor number, which was not needed in the symmetric case. As before, t and *tmax* are integers recording the current and the maximum times, respectively. The main loop has nine lines.

[1] *MPBegin*();

[2] **for** ($t = 0$; $t \leq tmax$; $t{+}{+}$)

[3] { **if** (*cproc* $\equiv 0$)
{ *converge* = *NewParameters*(*global*);
if (*converge*) **break**;
MPMasterSend(*global*, *W*); }

[4] **else** *MPSubordinateReceive*(*local*, *W*);

[5] *Simulate*(*local*);

[6] **if** (*cproc* $\equiv 0$) *MPMasterReceive*(*global*, *W*);
[7] **else** *MPSubordinateSend*(*local*, *W*); }

[8] *MPEnd*();

At the beginning of the loop, *cproc* is tested to determine whether the master processor is running (processor number 0). If so, then the master processor computes the new parameters, checks for convergence, and if convergence has not been achieved, sends the parameters out to all subordinates for additional computation (step 3 in the algorithm). If, on the other hand, a subordinate processor is running, it merely waits for the master processor to send it the parameters (step 4 in the algorithm).

After that, all processors, master and subordinate alike, run one simulation task by invoking function *Simulate* (step 5), as in the symmetric technique. Next the master processor receives the simulation results from all subordinates (step 6) while subordinates send them (step 7). Then the loop repeats. As before, W is the width of arrays $global$ and $local$.

In addition to the code for *MPBegin* and *MPEnd* (Functions 1 and 2 in the appendix) this process uses the twenty-eight lines of code of Functions 4a, 4b, 5a, and 5b in the appendix. The amount of data communicated is $2W(N-1)$ elements, where W is the number of elements per processor and N is the number of processors. Both master and subordinate processors pass messages twice per iteration.

3. Discussion

The symmetric version needs no conditional if–else statements to determine which processor is running. It makes half the number of calls to the message passing interface per iteration, which simplifies the communications. It is somewhat less error prone, not only because of its greater simplicity, but also because multiple processors can be automatically checking each others work, detecting such mistakes as uninitialized variables that may behave differently under different conditions. It is shorter. The symmetric version uses 5 lines within its loop in the algorithm above and calls upon the 5 lines of Function 3 in the appendix, for a total of 10 lines in the loop. Functions 4a, 4b, 5a, and 5b in the appendix do not exist in the symmetric version. The master–subordinate version, in contrast, uses 9 lines within its loop and calls upon the additional 28 lines of Functions 4a, 4b, 5a, and 5b in the appendix, for a total of 37 lines in the loop.

The code to support symmetric multiprocessing is thus almost four times as compact. This savings can become compounded in the application code, because that code does not have to differentiate between master and subordinate communications. In parts of the code not connected with inter-processor communication, such as printing, one processor may still have to act as master. Yet since each level of reduction in complexity can be significant in a large program, this technique is preferred, all other things being equal.

One thing not equal is energy consumption. With all processors computing during the contraction phase, more heat will be generated and more energy consumed. Computations performed during the contraction phase will typically be short and simple compared with long and complicated simulations in the expansion phase, so this will be negligible. But if it is not, then a master–subordinate approach might be preferred.

Another thing not equal is the amount of information communicated among processors. The master–subordinate technique has only $N-1$ communication paths, where N is the number of processors, whereas the symmetric technique has $\frac{1}{2}N(N-1)$ paths (Figure 1). Though this can be a large difference, it can also be insignificant in many applications. If the computation step is seconds or minutes or more, as it often will be, the microseconds or milliseconds dedicated to communications will vanish into the rest of the computation.

4. Conclusions

The symmetric version is simple. It is a viable way of communicating among multiple processors that can be incorporated into any expansion–contraction simulation programs, or related kinds of simulations. We have used it successfully in a large-scale simulation model developed by one of us (A.K.) for human tuberculosis in the UK. Compilable copies of the code described here and related simulation algorithms are available free from the authors upon request.

5. Acknowledgements

We are grateful to Shuxia Zhang and David Porter for helpful discussions of the message passing interface, to Mark Nelson for generous guidance to make things work, to Todd Lehman for reading the manuscript and helping with presentation of the code, and for Richard Barnes for help of many kinds.

6. Contributions

C.L. and A.K. worked together on various techniques for controlling the multiprocessors on a large-scale simulation carried out by A.K. C.L. conceived and coded the symmetric technique and A.K. built it into the simulation program and tested it in actual use. Both authors reviewed the code and contributed to the manuscript.

7. Funding

This project was supported in part by grants of computer time from the Minnesota Supercomputer Institute, Minneapolis, Minnesota, and by doctoral research funding to A.K. from the Modelling and Economics Unit at the Health Protection Agency, London.

References

[1] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, "MPI: The complete reference," *MIT Press, Cambridge, MA*, 2012.

8. Appendix

This appendix defines the precise connections with the "Message Passing Interface," MPI [1]. Prototypes for functions and constants are defined in a file "mpi.h", which must be included at the top of the code. Then each process must call Function 1 before beginning. That function initializes communications, determines the number of processors that are participating, and assigns a number to the current processor. Furthermore, each process calls Function 2 after its work is complete, just before ceasing operations.

Function 3 is used in both symmetric and master–subordinate techniques. In this example it assembles an *nproc* by *w* array of data from all processors, where *nproc* is the number of processors and *w* is the number of data elements shared by each processor. It is typically called at the end of each processing step to synchronize all processors and put them all in a common data state. Processing is delayed until all processors have called this function. Therefore, note that all processors must call at corresponding points in the cycle or operations could deadlock.

Functions 4a, 4b, 5a, and 5b are additional algorithms needed for master–subordinate processing. Function 4a sends data from the master processor, numbered 0, to all processors allocated, including itself. Data to be sent reside in the *nproc* by w array of data. Function 4a is typically called at the beginning of each processing step to synchronize all processors and give each processor the data it needs to carry out the next step. The master process must call this function and all others must call the companion function *MPSubordinateReceive*, Function 4b, at corresponding points in the cycle. Function 4b receives data from the master processor, resulting from that processor's call to 4a.

Function 5a receives data from subordinate processors, whose numbers are greater than 0. Data are assembled in the *nproc* by w array. Function 5a is typically called by the master processor at the end of each processing step to receive results back from all processors and compute the data to begin the next step. The master process must call this function and all others must call the companion function *MPSubordinateSend*, 5b, at corresponding points in the cycle. Function 5b sends data back to the master processor, numbered 0, to satisfy its call to *MPMasterReceive*, Function 5a.

Function 1.

Upon entry to the algorithm, no conditions are significant. **At exit, (1)** multiprocessing operations have commenced. **(2)** $nproc$ contains the total number of processors allocated. **(3)** $cproc$ contains the number of the current processor, in the range 0 to $nproc - 1$.

```
int MPBegin()
{ static int argc; static char **argv; int n;
  MPI_Init(&argc, &argv);                          1. Initialize message processing.
  MPI_Comm_rank(MPI_COMM_WORLD, &cproc);           2. Determine this processor's number.
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);           3. Determine the number of processors.
  return 0; }                                      4. Return to caller.
```

Function 2.

Upon entry to the algorithm, multiprocessing operations have closed. **At exit,** the main program may itself exit.

```
int MPEnd() { MPI_Finalize(); return 0; }
```

Function 3.

Upon entry to the algorithm, (1) $local$ is a vector of w data elements (double precision floating point) that are this processor's contribution to the global data set. **(2)** $global$ is a $nproc$ by w matrix to receive the values of $local$ from all processors. **(3)** w contains the width of $local$ and $global$. **At exit,** $global[n]$ contains a copy of the contents of $local$ from each processor n, where n ranges from 0 to $nproc - 1$. In particular, the $local$ of this processsor passed on entry is in row $global[cproc]$.

```
int MPCommon(double global[][], double local[], int w)
{ MPI_Allgather(local, w, MPI_DOUBLE,
                global, w, MPI_DOUBLE,
                MPI_COMM_WORLD);
  return 0; }
```

Function 4a.

Upon entry to the algorithm, (1) $cproc$ is 0. **(2)** $global$ is a $nproc$ by w matrix containing values to be sent to each processor n in row $global[n]$. **(3)** w contains the width of $global[i]$. **At exit,** $global[n]$ has been sent to each processor n.

```
int MPMasterSend(double *global, int w)
{ double *temp =
    (double*)malloc(w*sizeof(double));
  MPI_Scatter(global, w, MPI_DOUBLE,
              temp, w, MPI_DOUBLE,
              0, MPI_COMM_WORLD);
  free(temp);
  return 0; }
```

1. Allocate a temporary area to receive the master's data back from itself.
2. Send data from $global[n]$ to each processor n.
3. Release the temporary area.
4. return to caller.

Function 4b.

Upon entry to the algorithm, (1) $cproc$ is not 0. **(2)** $local$ is vector of w elements to receive data from the master processor. **At exit,** $local$ contains the data received.

```
int MPSubordinateReceive(double local[], int w)
{ MPI_Scatter((double*)0, 0, MPI_DOUBLE,
              local, w, MPI_DOUBLE,
              0, MPI_COMM_WORLD);
  return 0; }
```

Function 5a.

Upon entry to the algorithm, (1) $cproc$ is 0. **(2)** $global[nproc][w]$ is an area to receive values for each processor n in row $global[n]$. **(3)** $global[0]$ contains any results from the master processor, to be sent back to itself. **(4)** w contains the width of $global[i]$. **At exit,** $global[n]$ contains the results from each processor n, except that $global[0]$ is unchanged.

```
int MPMasterReceive(double global[][], int w)
{ int i;
  double *temp =
    (double*)malloc(w*sizeof(double));
  for (i = 0; i < w; i++) temp[i] = global[i];
  MPI_Gather(temp, w, MPI_DOUBLE,
             global, w, MPI_DOUBLE,
             0, MPI_COMM_WORLD);
  free(temp);
  return 0; }
```

1. Allocate a temporary area to receive the master's data back from itself.
2. Send data from $global[n]$ to each processor n.
3. Release the temporary area and return to caller.

Function 5b.

Upon entry to the algorithm, (1) $cproc$ is not 0. **(2)** $local$ is vector of w elements to send to the master processor. **At exit,** the data have been sent.

```
int MPSubordinateSend(double local[][], int w)
{ MPI_Gather(local, w, MPI_DOUBLE,
             (double*)0, 0, MPI_DOUBLE,
             0, MPI_COMM_WORLD);
  return 0; }
```

1. Send data to processor 0.
2. Return to caller.

Halo Gathering Scalability for Large Scale Multi-dimensional Sznajd Opinion Models Using Data Parallelism with GPUs

K.A. Hawick and D.P. Playne
Computer Science, Institute for Information and Mathematical Sciences,
Massey University, North Shore 102-904, Auckland, New Zealand
{k.a.hawick, d.p.playne }@massey.ac.nz; Tel: +64 9 414 0800 Fax: +64 9 441 8181

March 2012

ABSTRACT
The Sznajd model of opinion formation exhibits complex phase transitional and growth behaviour and can be studied with numerical simulations on a number of different network structures. Large system sizes and detailed statistical sampling of the model both require data-parallel computing to accelerate simulation performance. Data structures and computational performance issues are reported for simulations on single and multi-core processing devices. A discussion of optimal data structures for performance on Graphical Processing Units using NVIDIA's Compute Unified Device Architecture (CUDA) is also given. System size and memory layout tradeoffs for different processing devices are also presented.

KEY WORDS
Sznajd opinion formation model; complex system; simulation; data-parallelism; GPU; CUDA; memory halo

1 Introduction

Opinion formation models [21] such as the Sznajd model [40] display some interesting phase transitional and complex behaviours that rely upon computer simulation for their study. Opinion formation models can capture individual behaviour at a microscopic level that manifests itself as a macroscopic or system-wide outcome when implemented with system of many participating agents. The Sznajd model of opinion formation exhibits complex phase transitional and growth behaviour and can be studied with numerical simulations on a number of different network structures [18, 35].

One of the main points of interest in simulating this sort of model is to study the dynamics [29, 43] and kinetics [5, 11] of the system both from the perspective of a well-defined model of phase transitional behaviour but also for to study the spread of influence of opinions [13,22,44] for comparison with real sociological phenomena.

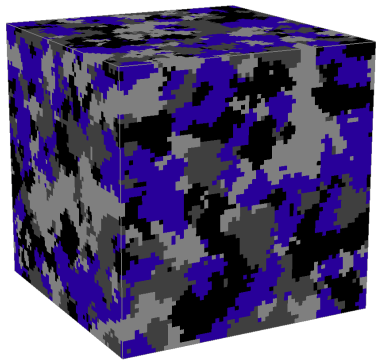


Figure 1: Multi-opinion Sznajd Model simulated on a 40^3 periodic lattice with 4 opinions in the population.

Sznajd models can be simulated on a range of network structures with varying number of neighbouring opinion sites [33]. Results have been reported for one dimensional systems [32]; triangular lattices [23] and generalised heterogeneous graphs and small-world [1] and Watts-Strogatz graph structures [42]. In this present paper we focus on square lattices but our simulation system can also manage arbitrary dimensional hyper-cubic structures in 3 dimensions and higher. Figure 1 is a system snapshot from our simulation, and shows the ongoing formation of spatial groups of like-minded opinion holding agents arranged on a cubic symmetry lattice.

This class of model can capture important sociological behaviours such as conformity [36]; crowd dynamics [25]; political extremism and relative agreements [9]; and voter herding and the role of independents [20]. Sznajd models can be usefully compared with real sociological systems such as political elections [34]; proportional and majority elections [41]; and other sociological and knowledge exchange scenarios [28].

The Sznajd model and its variants are also of interest as abstract models for the study of dynamics and criticality. Nu-

merical experiments and data obtained from simulations allow comparisons with theoretical predictions. An area of theoretical comparison is with non-equilibrium [10] and growth theories including Fokker-Planck theory, fluctuation dissipation theory [12] and other relaxation modelling equations [26].

Comparisons are drawn in the literature between the Sznajd model and related systems like the Ising model [37]. The Sznajd system has been described as a "push" model as influence is pushed outwards from an updated cell, whereas the Ising model is a "pull" model since it changes a cell's value based upon the value of its immediate neighbours [4, 39]. Other areas of comparison are based on the use of externally applied opinion or magnetic field biases [3, 7]. Active areas of development include incorporation of bulk parameters such as an effective temperature into Sznajd-like models [24, 38].

In summary, the Sznajd model and related models are an important class of complex system to study numerically. Large and fast simulations are necessary to support useful comparisons with realistic real systems and with theoretical predictions. We can achieve large fast simulations using data-parallel computing techniques.

Graphical Processing Units (GPUs) are finding many important uses as engines for data parallelism, over and above their original purpose for accelerating graphics systems. Opinion formation models such as the Sznajd system are relatively well suited to this form of parallel simulation and GPUs enable a rapid exploration of the parameter space of such models is possible interactively and with good statistical sampling on relatively large model systems.

Compute Unified Device Architecture (CUDA) [30] is NVIDIA's proprietary programming language that is widely used in programming computational model simulations including Ising systems. In this present paper we show how the Sznajd model can be implemented on GPU systems but that there are some unexpected complications due to the particular nature of the Sznajd model neighbourhoods. We also show that there is an interesting tradeoff between achieving realistically large system sizes the scalability requirements of simulating a system long enough to reach a well defined consensus opinion point.

Our paper is structured as follows: In Section 2 we summarise the Sznajd model and describe how we implemented it in Section 3. We include details on memory utilisation issues and a simple and more memory efficient implementation. We present some performance benchmark data on multiple GPU systems in Section 4. We offer some areas for further work in Section 5 and some conclusions in Section 6.

2 Sznajd Model Formulation

The Sznajd model formulation used in this research is formulated in the same manner as described in [18]. The simulation consists of some very simple microscopic rules that result in complex macroscopic behavior. Each update consists of selecting a cell and a random neighbor. If the two cells have the same opinion they convince their direct neighbors. This particular variation of the Sznajd model update can be applied for Sznajd systems of any dimensionality and is therefore useful for the parallel algorithmic and scalability analysis presented in this present work.

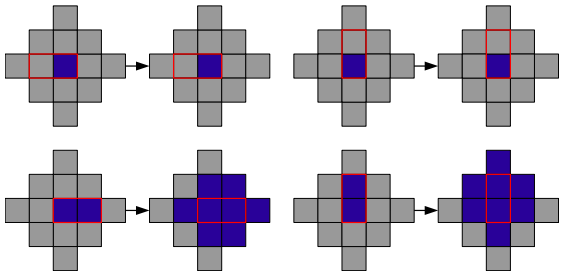


Figure 2: The update rules for the Sznajd Model, if both selected cells agree, they convince their neighbors. If they disagree the neighbors are unchanged.

The Sznajd model is constructed as follows:

1. each of the N lattice cells represents a voter agent holding a single opinion
2. starting conditions are chosen for a random mixture of Q different opinion states.
3. each voter is updated at each time step
4. upon choosing a voter, we randomly look at one of its neighbours.
5. if the voter and the chosen neighbour hold the same opinion they "persuade" all their immediate neighbours to this opinion
6. this process repeats until consensus is reached (whereby all voters hold the same final opinion state)

There are a number of properties that can be measured to study the changes in the population of opinion agents: the time to achieve consensus t_c; the time t_q to eliminate a species and yield only $q < Q - 1$ states present in the system; and the mean fluctuation sizes in these. These can be studied for different sizes of system N and also for different dimensionalities and indeed differing neighbour influence regions.

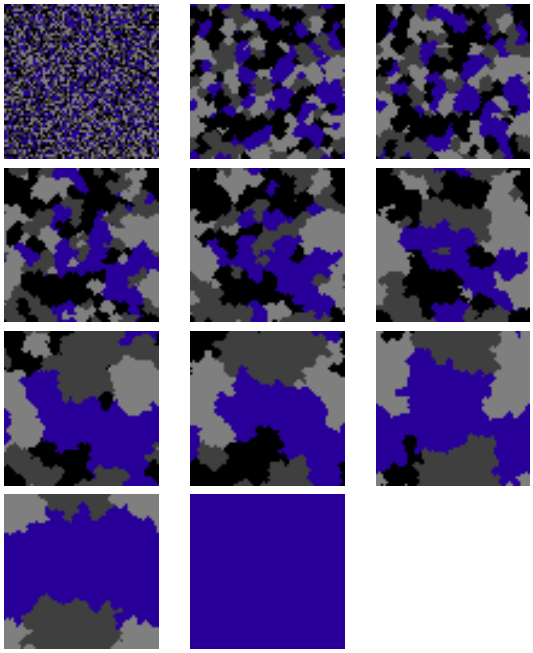


Figure 3: Sznajd Model on 64×64 periodic lattice at times: 0,1,2,4,8,16,32,64,128,256 and final configuration after random 25/25/25/25 start.

The Sznajd model and models like it can be run to full consensus under many dynamical conditions [29]. The meshes we study do lead (eventually) to a consensus outcome. The consensus state is a stable state since there are no thermal or spontaneous changes of opinion in the model we study. However although the consensus state will be arrived at eventually in finite time for a finite system, these completion times do grow with system size. We explore this effect and the implications for the maximum feasible system sizes that we can study.

Figure 2 shows the transition rules for the Sznajd model. The model behaviour is best understood from examination of a time series of model configurations. Figure 3 shows a two dimensional lattice of Sznajd opinion-holding agents evolving over logarithmic times from a random initial mix of opinions to a single consensus.

3 Data-Parallel Implementation

Sznajd model simulations present an challenge for Graphical Processing Units. In previous investigation of lattice-based simulations [15, 17, 19, 27, 31] GPUs generally exhibit the best performance for large system sizes as small simulations struggle to fully utilize the computational throughput of GPU architectures.

The nature of the Sznajd model means that the number of steps required for a system to reach consensus grows rapidly with system size. Because of this relationship, simulations of large system sizes that are well-suited to the GPU architectures require a very large number of steps to reach consensus. Even on powerful GPU architectures these large system sizes take an in feasibly long time to compute the thousands of runs required for statistical analysis.

This means an implementation must be developed that can perform effectively for small system sizes $N = 128^2$ to $N = 256^2$.

3.1 Sznajd Model Memory Access

The Sznajd model has a relatively large memory access pattern or memory halo. Each cell update has the potential to change the values of twelve different neighbouring cells. To perform parallel updates, it is important that these updates do not overlap or else the behavior of the model will be affected.

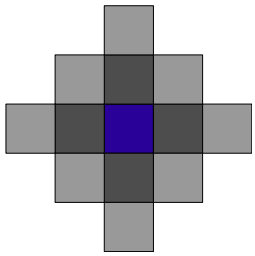


Figure 4: The potential memory halo of a Sznajd model update. The blue cell is the cell selected for update, the dark grey cells represent the immediate neighbors and the light grey cells represent cells whose values could potentially be affected.

3.2 Naive GPU Implementation

The initial idea of implementing the Sznajd model was to implement a checkerboard style update commonly used to simulate the Ising model [19]. The checkerboard update or red/black update performs a parallel update on every cell belonging to one half of the checkerboard. This update method is applicable to the Ising model because it reads from it's nearest neighbors and only changes the value of the cell being updated. Because the Sznajd model has quite a different memory halo, a different checkerboard pattern is required.

To ensure that no two parallel update ever write to the same cell, the memory halos of the updated cells must not overlap. One possible checkerboard design is to split the lattice into 5×5 (2D) or 5^3 (3D) sections. Each update selects the same cell out of this section and updates it. Each update will randomly select one neighbor but this is already accounted for in the memory halo. Importantly the

memory halos of the updates never overlap, this can be seen in Figure 5.

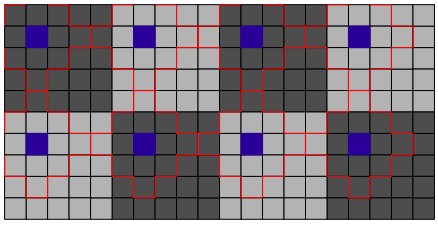


Figure 5: Update pattern of the naive GPU implementation, the update pattern ensures no cell value is changed by two different updates.

This implementation has regular memory access patterns and would be a perfect candidate for lattice crinkling which rearranges data in a lattice for better memory access [16]. However, during initial testing a problem was discovered with this implementation. When two neighboring cells are updated one after another, there is a tendency for an opinion to propagate in that direction. Because the cells are updated in a regular pattern, any neighboring updates will occur across the entire lattice. This means opinions propagate in one direction across the entire lattice and causes the lattice to 'jitter' during the course of the simulation.

Any correlation between updates such as this have the potential to affect the behavior of the model and skew any results. For this reason the Sznajd checkerboard update has not been used for any results gathering and instead an alternative implementation was developed to remove the correlation.

3.3 Improved GPU Implementation

The improved Sznajd update method is similar to the checkerboard update but with a few important differences. Instead of splitting the lattice into 5×5 sections, the improved method splits the lattice into 8×8 sections. These 8×8 sections are further split into 4×4 quadrants. Instead of the same cell in each section being updated at the same time, each quadrant of each section is updated at the same time. This update will randomly select a cell from the quadrant and update that cell. This breaks the correlation seen in the previous implementation while still insuring that the memory halos from two update never overlap. This update method is shown in Figure 6.

While this update method can be used to simulate the Sznajd model, it does severely restrict the computational load of each update. Each parallel update only processes one cell out of an entire 8x8 cell. This means for an N^2 system only $\frac{N^2}{8^2}$ cells can be updated in parallel. To perform an entire system update, 8^2 parallel updates are required.

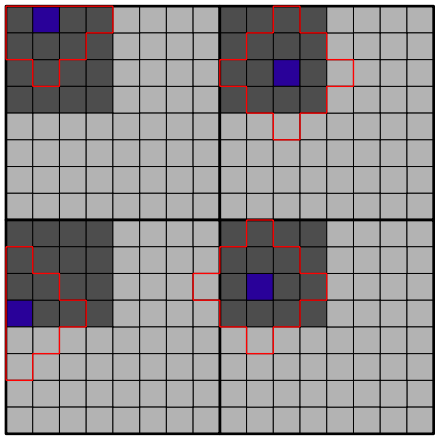


Figure 6: The update pattern of the improved GPU implementation. One random cell in each quadrant is updated

This severely restricts the performance of the simulation which is already limited to small system sizes due to the nature of the model but is necessary to break correlation between sequential updates.

3.4 Optimising the Implementation

Unfortunately due to the semi-random access pattern of this update method, previously successful optimization techniques such as data crinkling are not applicable [14, 19]. However, some optimization strategies such as bit-packing can be used. We are investigating Sznajd systems with $Q = \{2..16\}$, this means only 4 bits of storage are required for each cell. This allows the values of 8 cells to be packed into a single 32-bit integer. This conveniently allows each row of the 8x8 sections to be stored in a single integer.

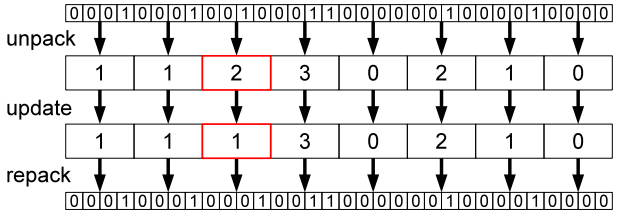


Figure 7: The process of unpacking integers, updating the simulation and re-packing them into the integers for storage.

This not only helps reduce the storage requirements for a Sznajd system but also helps memory access. Because each line of every 8x8 section will be stored in sequential memory addresses, they can be read in a single coalesced memory read. Once these values are read, they will be unpacked into a shared memory array which will be used to compute the simulation. The threads computing the up-

date will change the values in this shared memory array and then re-pack them into integers which will be written back to GPU memory. This process is illustrated in Figure 7.

4 Performance and Results

Due to the highly restricted and sparse update method coupled with very limited system sizes, the GPU implementation struggles to provide the kind of speed-ups seen in many simulations. For the system sizes appropriate for Sznajd simulations (64^2 - 256^2) the computational power of the GPU cannot be fully utilized. However, the GPU implementation can still performance faster than the CPU version for system sizes $N >= 160^2$.

As the system size increases, more of the computational power of the GPU can be used and the speedup it provides will increase. This performance improvement becomes increasingly important as the behavior of larger systems is investigated.

The performance of the GPU implementation has been compared to the un-optimised version and a CPU version for a range of system sizes and number of opinions. Varying the number of opinions had little impact on the performance of any simulation and only the results for $Q = 2$ are presented.

The CPU implementation has been written in C, compiled with gcc 4.5 and executed on a 2.67GHz Intel Core i7 920. Both the un-optimised (GPU1) and optimised (GPU2) implementations have been compiled using CUDA 4.0 and executed on a GTX580. The performance results of these three Sznajd model simulations are shown in Figure 8.

The improved performance of the GPU2 implementation allows larger system sizes to be investigated. The number of steps to consensus of the Sznajd model has been investigated for system sizes of $N = \{64^2...256^2\}$ and $Q = \{2...10\}$. The results of this experiment are presented in Figure 9 and appear to shown some unexpected results.

Previous work with system sizes of $N <= 96^2$ has shown that the number of steps required to reach consensus increases as the number of opinions increases [18]. However, the findings from the previous experiment shows that for Sznajd systems of $N >= 192^2$ and $Q = 2$ require more steps to reach consensus than systems with higher values of Q.

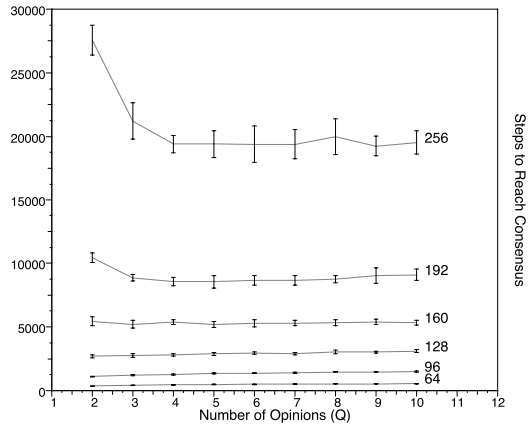


Figure 9: The number of simulation steps required for the Sznajd system to reach consensus.

5 Discussion

There are a number of variations to the simple Sznajd model that have been explored in the literature. The notion that two voters with the same opinion will have a strong influence on their immediate neighbours is not unreasonable, although it is possible to vary the number of agreeing voters required to cause a local sway of opinion. Triples and plaquettes of four have been tried. It appears that a pair is enough to capture the essential model behaviour however, and we have used the simplest "voter pairs" in this paper.

A simple regular mesh is not particularly realistic and a number of other meshes and graph networks including preferential and scale free network structures have also been studied and reported in the literature. Again, it appears that the square mesh captures the essence of the model behaviour and we use a simple square mesh in the work reported here. However we do consider the neighbourhood of influence of the agreeing pair of voters.

There are various ways the Sznajd model could be extended. One interesting possibility is modelling opinions that are continuous rather than discrete [2]. This sort of simulation would require use of floating point capabilities, which present a different set of performance tradeoffs on GPUs, which typically share floating point units across the their cores.

Another area for development is the study of agent reputations and how the reputation - or long term behaviour of an agent - affects its power of influence on its neighbouring agents [6, 8]. Temporal memory effects like reputation will require storage of agent histories and will considerably change the memory requirements. This will lower the feasibility of containing the simulation state within cache.

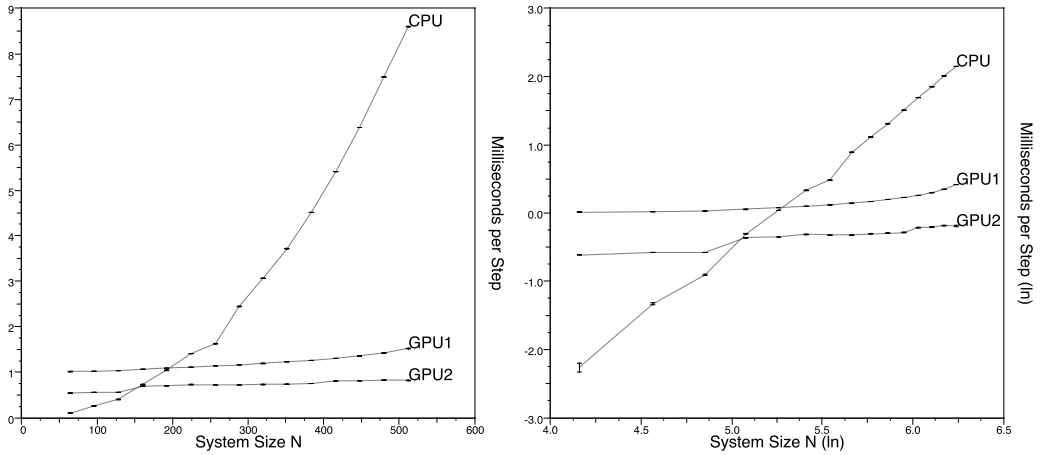


Figure 8: The milliseconds per time-step of three Sznajd model simulations - the CPU, GPU1 (un-optimised) and GPU2 (optimised) implementations. The results are shown in normal scale (left) and log-log scale (right).

6 Conclusion

We have discussed the importance of Sznajd opinion models and their role in simulation studies of real sociological systems and as models of phase transitional and complex emergent behaviour. We have described the need for large simulations that must be run to the point of single opinion consensus.

We have experimented with various data and multi parallel implementations and have identified a tradeoff that makes good use for multiple GPU devices attached to a CPU. The computational performance tradeoff space is not trivial and is dominated by the halo gathering scalability effects.

The Sznajd model neighbour-hood and the push nature of the information propagation means that the simulated system has to be arranged with a relatively large crinkle length, to safely delegate compute responsibility across the data parallel cores of a GPU. While this supports good scalability for large simulated system sizes, we are constrained to having systems sizes that are not so large as to make it infeasible to achieve single opinion consensus. Numerical experiments require us to run the simulations to this point for sensible comparisons with theory. The multi-GPU approach addresses this by achieving good job throughput.

Some areas for further development of Sznajd-like models have also been identified. These include the use of continuous opinion dynamics and historical reputation effects that would further exercise the floating point and memory management properties of GPUs.

References

[1] Bagnoli, F., Barnabei, G., Rechtman, R.: Small-world bifurcations in an opinion model. Tech. Rep. arXiv:0909.0117v3, University of Florence (December 2009)

[2] Biswas, S.: Mean field solutions of kinetic exchange opinion models. Phys. Rev. E 84, 056106 (2011)

[3] Candia, J.: Advertising and irreversible opinion spreading in complex social networks. Int. J. Modern Physics C 20, 1–20 (2009)

[4] Castellano, C., Pastor-Satorras, R.: Irrelevance of information outflow in opinion dynamics models. Phys. Rev. E 83, 016113 (2011)

[5] Chowdhury, K.R., Ghosh, A., Biswas, S., Chakrabarti, B.K.: Kinetic exchange opinion model: solution in the single parameter map limit. arXiv 1112.5328v1, Saha Institute of Nuclear Physics, Kolkata, India (December 2011)

[6] Crokadakis, N., Forgerini, F.L.: Competition among reputations in the 2d sznajd model: Spontaneous emergence of democratic states. Tech. rep., Universidade Federal Fluminense, Rio de Janeiro, Brazil (2011)

[7] Crokidakis, N.: Effects of mass media on opinion spreading in the sznajd sociophysics model. Physica A; Stat. Mech and its Applications 391, 1729–1734 (2012)

[8] Crokidakis, N., de Oliveira, P.M.C.: The sznajd model with limited persuasion: competition between high-reputation and hesitant agents. J. Stat. Mech: Theory and Experiment Online, P11004 (2011)

[9] Deffuant, G., Amblard, F., Weisbuch, G., Faure, T.: How can extremism prevail? a study based on the relative agreement interaction model. Journal of Artificial Societies and Social Simulation 5(4), 1–27 (2002), http://jasss.soc.surrey.ac.uk/5/4/1.html

[10] Descalzi, O., Marti, A.C., Masoller, C., Rosso, O.A.: Topics on non-equilibrium statistical mechanics and nonlinear

physics. Phil. Trans. Roy. Soc. A 367, 3151–3156 (2009)
[11] During, B.: Kinetic modelling of opinion leadership. SIAM News December, 1–12 (2011)
[12] During, B., Markowich, P., Pietschmann, J.F., marie Therese Wolfram: Boltzmann and fokker-planck equations modelling opinion formation in the presence of strong leaders. Proc. Roy. Soc A 465(2112), 3687–3708 (September 2009)
[13] Fortunato, S.: Damage spreading and opinion dynamics on scale free networks. Physica A: Statistical mechanics and Its Applications 348, 683–690 (2005)
[14] Hawick, K.A., Playne, D.P.: Automatically Generating Efficient Simulation Codes on GPUs from Partial Differential Equations. Tech. Rep. CSTN-087, Computer Science, Massey University (2010), submitted to Elsevier, J. Comp. Sci.
[15] Hawick, K.A., Playne, D.P.: Modelling, Simulating and Visualizing the Cahn-Hilliard-Cook Field Equation. International Journal of Computer Aided Engineering and Technology (IJCAET) 2(CSTN-075), 78–93 (2010)
[16] Hawick, K.A., Playne, D.P.: Hypercubic Storage Layout and Transforms in Arbitrary Dimensions using GPUs and CUDA. Concurrency and Computation: Practice and Experience 23(10), 1027–1050 (July 2011)
[17] Hawick, K.A., Playne, D.P.: Numerical Simulation of the Complex Ginzburg-Landau Equation on GPUs with CUDA. In: Proc. IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN). pp. 39–45. No. CSTN-070, IASTED, Innsbruck, Austria (15-17 February 2011)
[18] Hawick, K.: Multi-party and spatial influence effects on opinion formation models. In: Proc. IASTED International Conference on Modelling and Simulation (MS 2010). No. CSTN-032, Calgary, Canada (June 2010), paper 696-035
[19] Hawick, K., Leist, A., Playne, D.: Regular Lattice and Small-World Spin Model Simulations using CUDA and GPUs. Int. J. Parallel Prog. 39(CSTN-093), 183–201 (2011)
[20] Hisakado, M., Mori, S.: Digital herders and phase transition in a voting model. J. Phys. A: Math. and Theor. 44(27), 275204 (July 2011)
[21] Kacperski, K., Holyst, J.A.: Opinion formation model with strong leader and external impact: A mean field approach (1999)
[22] Kempe, D., Kleinberg, J., Tardos, E.: Maximizing the spread of influence through a social network. In: Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data MiningKDD'03 (2003)
[23] Kondrat, G., Sznajd-Weron, K.: Percolation framework in Ising-spin relaxation. Phys. Rev. E 79, 011119–1–5 (2009)
[24] Kondrat, G.: How to introduce temperature to the 1 dimensional sznajd model? Physica A: Stat. Mech and its Applications 390, 2087–2095 (2011)
[25] Koster, G., Seitz, M., Treml, F., Hartmann, D., Klein, W.: On modelling the influence of group formations in a crowd. Contemporary Social Science 6(3), 397–414 (November 2011)
[26] de la Lama, M., Szendro, I.G., Iglesias, J.R.: Van kampen's expansion approach in an opinion formation model. Eur. Phys. J. B 51, 435–442 (2006)
[27] Leist, A., Playne, D., Hawick, K.: Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. Concurrency and Computation: Practice and Experience 21, 2400–2437 (December 2009), CSTN-065
[28] Lella, L., Licata, I.: A new model for the organizational knowledge life cycle. In: Minati, G., Abram, M., Pessa, E. (eds.) Processes of Emergence of Systems and Systemic Properties, pp. 215–228. Springer (2007)
[29] Martins, A.C.R.: Bayesian updating rules in continuous opinion dynamics models. J. Stat. Mech. P02017(arXiv:0807.4972v1), 1–14 (2009)
[30] NVIDIA® Corporation: CUDA™ 2.1 Programming Guide (2009), `http://www.nvidia.com/`, last accessed May 2009
[31] Playne, D., Hawick, K.: Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA. In: Proc. 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09) Las Vegas, USA. No. CSTN-073 (13-16 July 2009)
[32] Przybyala, P., Sznajd-Weron, K., Tabiszewski, M.: Exit probability in a one-dimensional nonlinear q-voter model. Phys. Rev. E 84, 031117 (2011)
[33] Schwammle, V., Gonzalez, M.C., Moreira, A.A., Jr. Andrade, J.S., J., H.H.: The spread of opinions in a model with different topologies. Phys. Rev. E 75, 066108 (2007)
[34] Sobkowicz, P.: Modelling opinion formation with physics tools: Call for closer link with reality. Journal of Artificial Societies and Social Simulation 12(1), 1–11 (January 2009)
[35] Staffer, D.: Monte carlo simulations of sznajd models. Journal of Artificial Societies and Social Simulation 5(1), 1–9 (January 2001)
[36] Sznajd-Weron, K., Tabiszewski, M., Timpanaro, A.M.: Phase transition in the sznajd model with independence. Europhysics Letters 96, 48002 (2011)
[37] Sznajd-Weron, K.: Dynamical model of ising spins. Phys. Rev. E 70, 037104–1–4 (2004)
[38] Sznajd-Weron, K.: Sznajd model and its applications. Acta Physica Polonica B 36(8), 2537–2547 (2005)
[39] Sznajd-Weron, K., Krupa, S.: Inflow versus outflow zero-temperature dynamics in one dimensional. Phys. Rev E 74(74), 031109–1–8 (2006)
[40] Sznajd-Weron, K., Sznajd-Weron, J.: Opinion evolution in closed community. Int. J. Modern Physics C 11(6), 1157–1165 (2000)
[41] Timpanaro, A.M., Prado, C.P.C.: Generalized sznajd model for opinion propagation. Phys. Rev. Lett. 80(2), 021119 (2009)
[42] Timpanaro, A.M., Prado, C.P.C.: Coexistence of interacting opinions in a generalized sznajd model. Phys. Rev. E 84, 027101 (2011)
[43] Woloszyn, M., Stauffer, D., Kulakowski, K.: Phase transitions in nowak-sznajd opinion dynamics. Physica A 378, 453–458 (2007)
[44] Xu, X.J., Wu, Z.X., Chen, G.: Epidemic spreading in lattice-embedded scale-free networks. Physica A 377, 125–130 (2007)

Ultra-high resolution atmospheric global circulation model NICAM on graphics processing unit

I. Demeshko[1], **S. Matsuoka**[2], **N. Maruyama** [3], **and H. Tomita** [3]
[1]Dep. of Mathematical and Computing Science, Tokyo Institute of Technology, Tokyo, Japan
[2]Global Scientific Information and Computing Center, Tokyo Institute of Technology, Tokyo, Japan
[3]RIKEN Advanced Institute for Computational Science, Kobe, Japan

.

Abstract— *Climate simulations have significant role in analyzing changes that have occurred in the Earth, give us better understanding of the recently happened processes. Most of the existing models perform approximate results and fundamental improvements of the software models are necessary to increase accuracy. We present an efficient implementation of ultra-high resolution atmospheric global circulation model on graphics processing units(GPUs). The model based on the nonhydrostatic system with the icosahedral grid and called "NICAM". We ported computationally intensive part of the NICAM code to GPU by using CUDA FORTRAN, then validated and compared its GPU performance to that for the parallel CPU version of the code. This approach shows a good performance results together with reducing memory consumption compared to a fully GPU approaches. Our results show 5x speedup for a computationally intensive part of shallow water simulation model on a single GPU in comparison with a parallel CPU implementation (5 cores).*

Keywords: GPU computations, CUDA Fortran, climate simulations, nonhydrostatic system

1. Introduction

One of the main goals of climate simulations is a prediction of future climate changes and their impact on the Earth and society. It is essential to get a reliable results from the simulations to plan our future ecological, financial and human strategy. For this purpose we need significantly increase scales of the current climate simulations. Thus, it is important to adapt current climate applications to take advantage of the performance capabilities of novel hybrid architectures.

Performance rise of the climate and weather simulation software was based on increasing processor speed rather than increasing parallelism for last years[1]. Nowadays, to sustain such an increase towards the exascale era, we need some significant changes in the software models. GPUs are a very high performance alternative to conventional microprocessor. The massive parallelism of GPUs offers tremendous performance in many high-performance computing applications. GPUs were designed to exploit fine-grained parallelism, which gives us an ability to create weather and climate models with much finer parallelism.

In order to increase performance of the existing climate simulation, we ported and optimized a high resolution climate model to GPU. We aimed to get significant acceleration from applying heterogeneous computing with both conventional CPUs and vector-oriented GPU accelerators.

In this work we use single GPU to run a new type of ultra-high resolution atmospheric global circulation model NICAM (Nonhydrostatic ICosahedral Atmospheric Model) [2] being developed at Advanced Institute for Computational Science, RIKEN (see section II). Initial NICAM code uses 2-dimensional MPI-parallelization and shows good scalability results.

We propose to localize and port the most computation-intensive part of the climate model to GPU. For this purpose, we first study the original NICAM code and isolate the most time-consuming modules. Then, we ported those modules to GPU and evaluated the performance of our implementation.

The main difference of our strategy from some of existing GPU-based approaches [3], [4] is that we do not port to the GPU entire climate model application. Our method allows to reduce the memory to be allocated on GPU by performing on CPU some low-cost computations and porting to the GPU only the most time-consuming computations.

The results of our evaluation show that we reach an almost optimal performance for most of the ported kernels and we see an important speedup. We have got 5x speedup for a computationally intensive part of shallow water simulation model on a single GPU, in comparison with a parallel CPU implementation. Our GPU kernels give us performance, close to the maximum one, which indicate the efficient hardware utilization.

This paper is organized as follows. We describe some important background in the "NICAMmodel" section. Section III outlines our GPU NICAM implementation approach, presents step-by-step algorithm of the "mapping to GPU" process. In section IV we give some information about environment we used, describe performance results for both CPU and GPU versions of the code and present GPU performance model. We give some conclusions and outline our plans for future work in section V.

2. NICAM model

NICAM is a Nonhydrostatic ICosahedral Atmospheric Model, used as a Global Cloud Resolving Model (GCRM). It was designed to perform "cloud resolving simulations" by directly calculating deep convection and meso-scale

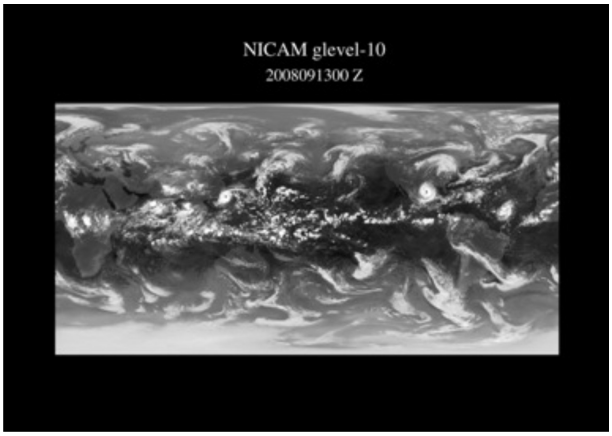


Fig. 1: Tropical cyclones SINLAKU and IKE reproduced by NICAM 7km simulation [6] (00UTC 13 Sep. 2008) (by H.L. Tanaka).

circulations, which play key roles not only in the tropical circulations but in the global circulations of the atmosphere[2]. NICAM - is a unified model in the sense that it can be used for both short term numerical predictions for weather systems such as a week, and long term simulations to obtain quasi-equilibrium climate states (see Figure 1).

The model uses fully compressible (elastic) nonhydrostatic system to obtain thermodynamically quasi-equilibrium states by long time simulations. For this purpose a nonhydrostatic numerical scheme which guarantees conservation of mass and energy was devised and implemented to the global model using the icosahedral grid configuration. The formulation and numerical scheme of NICAM along with numerical results of some test cases are thoroughly presented in [5].

The finite volume method is used for numerical discretization, so that total mass and energy over the domain is conserved; thus this model is suitable for long term climate simulation.

2.1 Numerical methods

For the horizontal discretization, icosahedral grid system on the sphere is used. Figure 2 shows an example of series of consecutive icosahedral grids.

The icosahedral grids are constructed by a recursive division of geodesic arches on the sphere. Starting from the original icosahedron, one-level finer grids are generated by bisecting the geodesic arches of the former coarser grids. We call the n-th bisection of the icosahedron glevel n (glevel: grid division level). The average grid interval of glevel 11 is about 3.5 km, for example. The total number of grid points is Ng = $10(2^n)^2 + 2$.

Numerical models with this grid system are first investigated by Sadourny et al. [7] and Williamson [8], and are recently revisited as a candidate for next-generation high-resolution global models.

In the [9] Tomita et al. describe the modifications, which was applied to the original icosahedral grids by using the

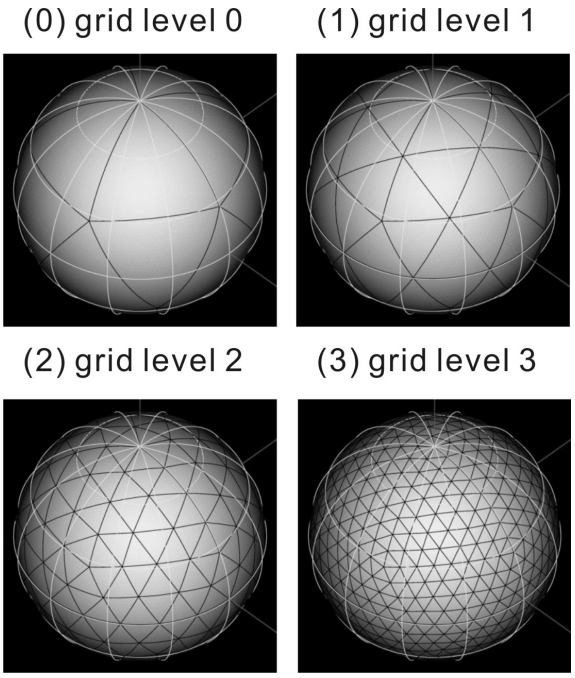


Fig. 2: Horizontal discretization scheme of NICAM model, based on icosahedral grid system on the sphere. Grid level 0 - original icosahedron, 2 grid points. The number of grid points for grid level n can be calculated by $Ng = 10(2^n)^2 + 2$

spring dynamics. With this modifications, the fractal structure of the original icosahedral grid is relaxed and more uniform grid structure with a smaller ratio of minimum to maximum grid intervals is obtained by smoothing the grid arrangement. It was found that the numerical errors can be reduced using the modified grid[10].

The governing equations of the global model are a newly developed nonhydrostatic schemes that guarantees conservation of mass and energy. The finite volume method is used for the flux form equations. The Arakawa-A type grid is used where all the variables are allocated at the vertices of triangles. The shape of the control volume is either hexagon or pentagon.

Further details about numerical scheme, used in NICAM are described in [2], [9], [10].

2.2 MPI parallelization

In this paper we present our single GPU implementation of NICAM model, but we plan to use current NICAM MPI-parallelization method to create a multi-GPU implementation for a following work.

Initial NICAM code use 2-d domain decomposition with FLAT-MPI parallel programming model (see Figure 3, Figure 4).

MPI parallelization strategy is based on discretization grid by regions, which then managed by the different MPI processes.

Region discretization algorithm is shown at the Figure 3. First, we create region level 0 by connecting two neighboring icosahedral triangles. In this case we have

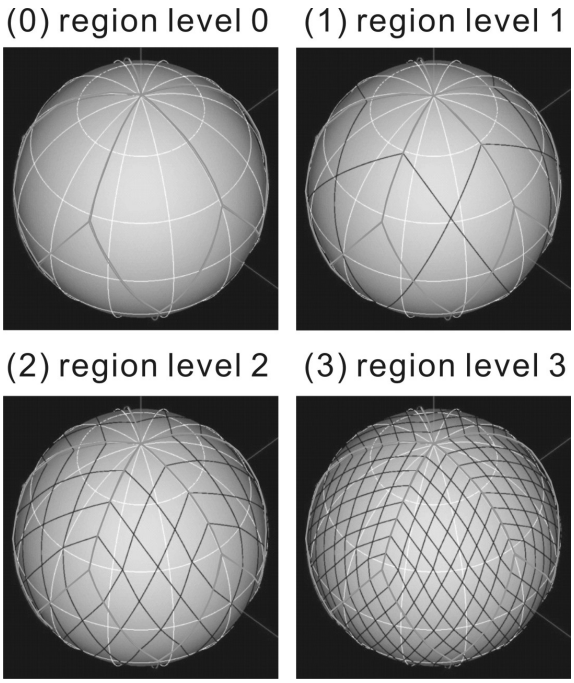


Fig. 3: NICAM region decomposition, used for FLAT MPI parallelization scheme. Region level 0 has 10 rectangles. The number of rectangles for the region level n can be calculated by $Nr = 10(4^n)$

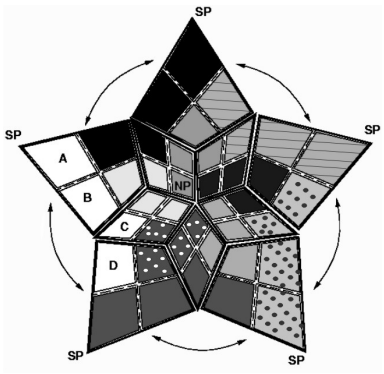


Fig. 4: MPI distribution strategy: region level 1, 40 regions and 10 MPI processes. Each process manage 4 regions with the same color

only 10 rectangles for the level 0 region. To increase region level to 1, we divide each of rectangles into 4 sub-rectangles by connecting the diagonal mid-points (revel-1). Continuing this process recursively we can get region level-n.

MPI distribution strategy is presented on Figure 4. One process manages rectangle regions with the same color. Figure 4 shows an example case for the region level 1, 40 regions and 10 MPI processes. Each process manages 4 regions with the same color.

Assuming one process manages one rectangle region, increasing r-level computational intensity on 1 process is reducing.

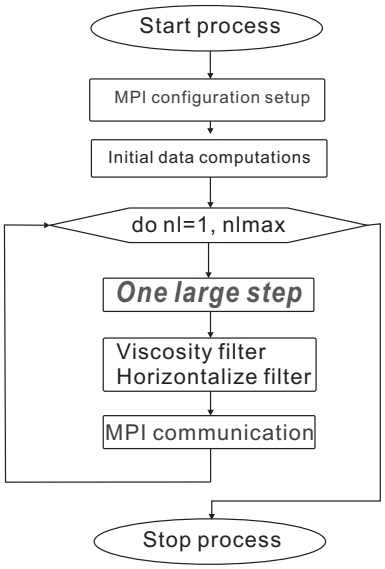


Fig. 5: Simplified flow-graph of the original NICAM Shallow Water model code.

3. GPU implementation

Our GPU implementation approach is based on porting the most computationally intensive part of the initial NICAM code to GPU. In this work we investigate 2-dimensional (Shallow water) case of NICAM model [11], which plan to implement for the 3- dimensional case in future work.

First, we selected a computationally intensive module of NICAM, then modified this Fortran model to run on NVIDIA GPU using PGI CUDA Fortran[12], [13], validated and compared its GPU performance to that of the original MPI parallel module.

In purpose to analyze runtime behavior of the given code we used the Scalasca performance toolset [14]. We have got profiling results for different configurations of the grid and region level numbers.

In the Figure 5 we present a simplified flow-graph of the original NICAM Shallow Water model code. The main computations performs in the cycle by the time steps (nl). Before starting the main cycle computations we need to setup MPI configuration and compute the initial data. At the end of each time step we collect the data from each MPI process and go to the next time iteration.

According to the profiling results "one large step" is the most time consuming module of the code. It takes more then 50% of the whole time to compute this module. Therefore, in purpose to accelerate computations, we decided to port this module to GPU.

In the Figure 6 we present our GPU implementation approach scheme.

After we computed an initial data on CPU we send the ones, necessary for "One large step" computations, to GPU and store them in GPU global memory. Some of this initial data are constant and by porting them once in the beginning of the NICAM computations we reduce CPU-GPU communicational overheads.

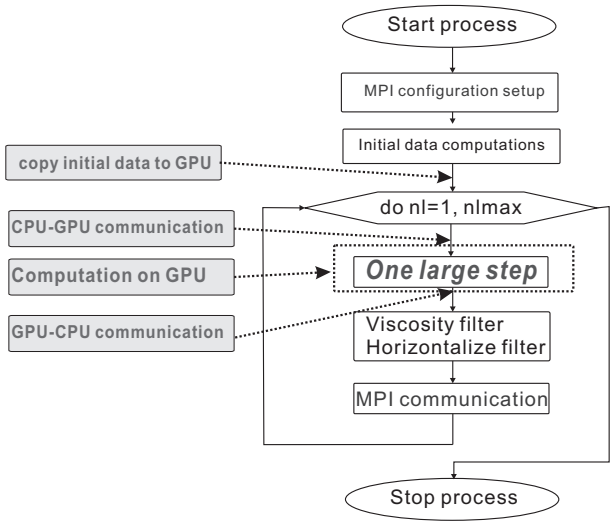


Fig. 6: NICAM GPU implementation approach. Porting the most computationally intensive module "One large step" to the GPU

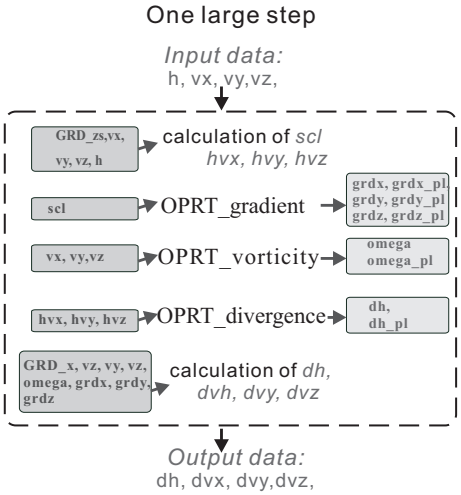


Fig. 7: "One large step" module. Left side blocks - input data, right side blocks - output data.

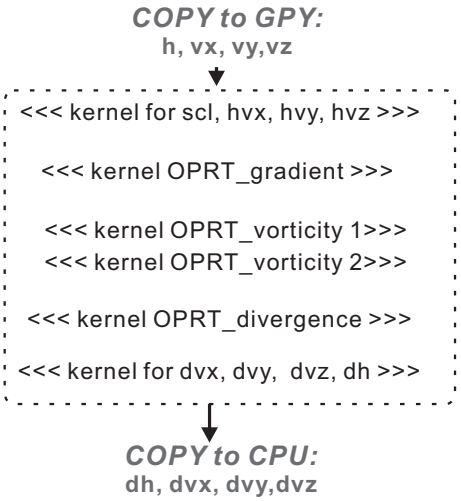


Fig. 8: "One large step" on GPU.

The data, which are variable inside the cycle and necessary for the GPU computations, are transferring to GPU before the "One large step" module on each time step. Then, data, computed on GPU and needed for farther computations on CPU, copying to the CPU at the end of the "One large step".

In the Figure 7 we present "One large step" module in detail. This module performs one full time step of the second-order Runge-Kutta scheme, which uses for the approximation of solutions of ordinary differential equations. "One large step" module consist of 3 main subroutines: OPRT_gradient, OPRT_vorticity and OPRT_divergence. If we port only this subroutines to GPU the communication overheads will be significant and affect to the performance. Therefore we also port initial data and output data computation modules to GPU and keep all temporary data in the GPU global memory. Initial CPU code for this module was slightly modified in purpose to reduce the amount of memory to be allocated on GPU.

We created one kernel for OPRT_gradient and one for OPRT_divergence. For OPRT_vorticity module it was necessary to create 2 kernels for the data synchronization issue. Figure 8 shows how our numerical scheme is computed by executing 6 CUDA kernels in order.

Each module, ported to the kernels, based on a nested loop calculation. Each loop iteration computes 1 element of 2-dimensional array. Each thread of our GPU kernels calculates 1 element of the array.

The CUDA programming model requires the programmer to organize parallel kernels into a grid blocks, which divided into thread blocks with at most 512 threads each. The NVIDIA GPU architecture executes the threads of a block in SIMT (single instruction, multiple thread) groups of 32 called warps.

We use 2-dimensional grid, which size depends on the of grid level and region level sizes. We use a block configuration of 256 threads where we have one thread per element. Our block size is a multiple of 32 which fits with the warp size and, therefore, allow us to achieve maximum efficiency.

We have h, vx, vy and vz arrays as an input data for the GPU computations, and dh, dvx, dvy, dvz as an output data. All the data, used for GPU computations, are stored in GPU Global memory during the entire NICAM model computation process. This allows different kernels access to the common data.

4. Evaluation

In this section we present results of implementation NICAM code on 1 node and single GPU.

4.1 Evaluation environment

Described approach was implemented on the TSUBAME 2.0 supercomputer, established at Tokyo Institute of Technology.

TSUBAME 2.0 consists of 1408 compute nodes (thin nodes) of two Intel Xeon Westreme-EP 2.9 GHz CPUs and tree NVIDIA M2050 GPUs with 52Gb and 3Gb of system

and GPU memory, running SUSE Linux Enterprise Server II SP1.

1 node has 2 sockets, 12 cores/node. Each node is interconnected by dual QDR Infinitband network with a full bisection- bandwidth fat-tree topology.

NICAM code consists of both FORTRAN and C++ modules. We use PGI CUDA FORTRAN $pgfortran$ version 2011 compiler for the GPU and FORTRAN code and $mpicc$ compiler for the C++ code.

4.2 Performance results

In purpose to compare NICAM CPU version performance with one for the GPU version we investigated initial CPU version for the code performance on TSUBAME 2.0 supercomputer. After performing the CPU code analysis we modified the code, porting the most time-consuming module to the single GPU, verified and validated our implementation. Then we compared MPI-parallel code performance with the single GPU code performance.

4.2.1 CPU performance

As it was described in section II, NICAM CPU code is based on 2d-domain decomposition with FLAT-MPI parallel programming model. Initial grid is divided by regions, managed by different MPI processes. The number of cores is restricted by the region division level, which can has limited number of the regions, that is $10*(2^n)^2$, where n - the region level order. Therefore, the total number of processes can be only the divisor of the total number of regions.

Figure 9 shows strong scalability results for the initial CPU version on TSUBAME 2.0 supercomputer. Problem size is grid level 11 with region level 5, which corresponds to 41943042 grid points. Strong scalability results demonstrate good speed up resulting on the number of cores up to 2560.

Figure 10 and Figure 11 present CPU performance results for the most time-consuming modules, ported to GPU.

From the CPU performance graph we can see that increasing grid level the scalability is rising due to increasing of the computational intensity per one process.

We assume 5 processes is a saturation point of the MPI parallelization and further compare CPU performance results on 5 processes with the GPU performance results on single GPU (see Figure 12)

4.2.2 GPU numerical performance

We compared single GPU version performance result with the performance for parallel MPI-parallel version (saturation point). We assume that the saturation point for performance of the MPI-parallel version achieves when the number of cores equal 5. Due to the fact that we have 6 cores in one socket on Intel Xeon X5670, we compare 1 CPU socket results versus 1 GPU socket.

This comparison results for the 3 main subroutines of the NICAM Shallow Water model are shown on the Figure 12. We can see that for the grid level 6 (gl06)

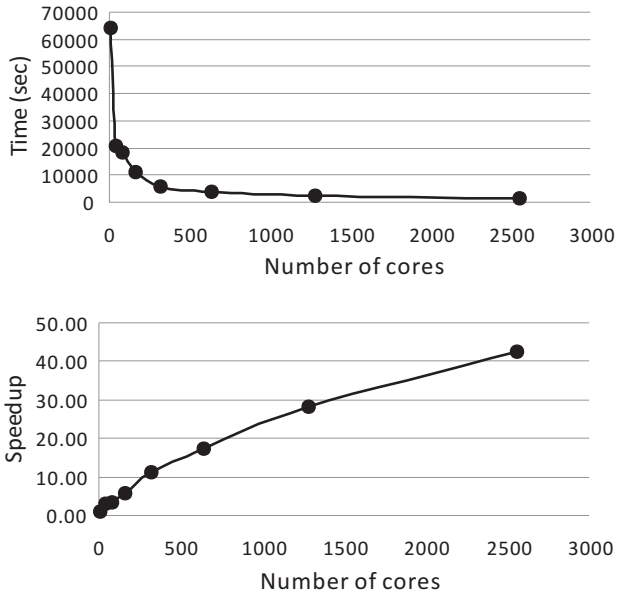


Fig. 9: Strong scalability of the CPU version. Average running time and speedup as a function of the number of cores for the grid level 11 and region level 5, which is correspond to horizontal size of grid cell around 3.5 km, 41943042 grid points

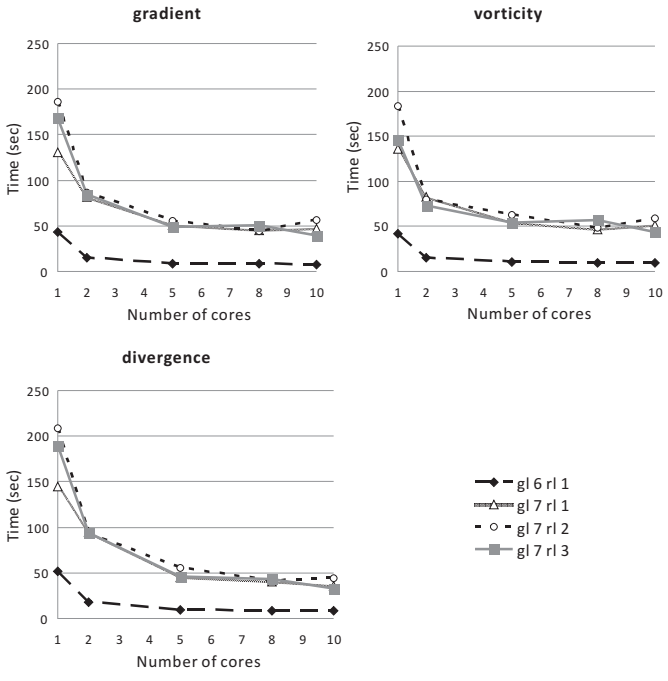


Fig. 10: CPU performance graph. Average running time as a function of the number of cores for the different configurations of the grid and region levels

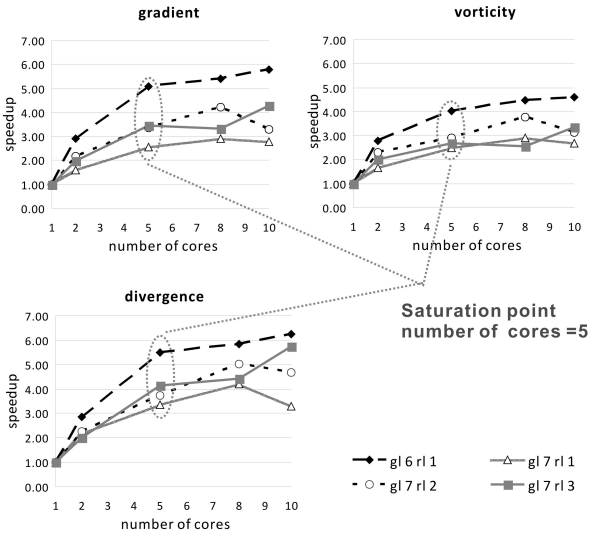


Fig. 11: CPU performance graph. Average speedup as a function of the number of cores for the different configurations of the grid and region levels

GPU versions performs 3 times faster then multi-CPU one. Increasing grid level to gl07 GPU version performs about 5 times faster then parallel-CPU one. It can be explained by the increasing computational intensity per kernel. We get 5x acceleration also for the grid level 8.

Due to limitations in GPU global memory, we were unable to scale NICAM code to a grid level higher then 8. The NVIDIA M2050 GPU only provides 3 GByte of Global memory. This limitation can be relaxed by using GPUs with higher amount of global memory. Also, as a future work, we are going to reduce amount of memory to be allocated per one GPU by splitting kernels between multiple GPUs, sharing 1 node.

4.2.3 GPU performance model

In purpose to measure the performance of our GPU NICAM implementation, we used "roofline" model of Samuel Willams [15]. This model compares achieved performance to a "roofline" graph of peak data streaming bandwidth and peak FLOP/s capacity.

We calculated performance by the next formula (1):

$$Performance = \frac{FLOP}{\frac{FLOP}{Fpeak} + \frac{Byte}{Bpeak} + \alpha} = \\ = \frac{FLOP/Byte}{\frac{FLOP}{Byte} + \frac{Fpeak}{Bpeak} + \alpha\frac{Fpeak}{Byte}} Fpeak \qquad (1)$$

Here $FLOP$ - number of floating-point operations for applications, $Byte$ - byte number of memory access for applications, $Fpeak$ - peak performance of floating-point operation, $Bpeak$ - peak memory bandwidth, α - time taken by other OPs except both FP and memory access.

For TSUBAME 2.0 supercomputer $Fpeak = 515GFlops$, $Bpeak = 148GByte/sec$ in double precision, we assume $\alpha = 0$.

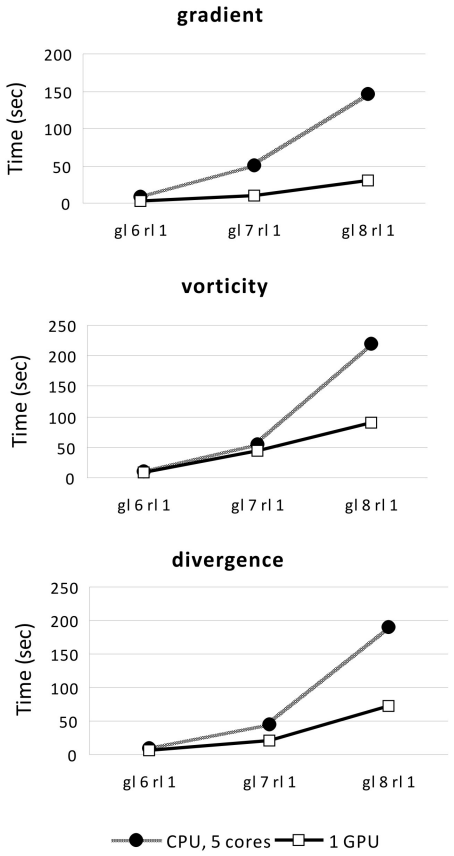


Fig. 12: Average running time for MPI-parallel CPU module versus time for GPU kernel as a function of the problem configuration for 3 main modules of the NICAM code.

Arithmetic intensity was calculated as $FLOP/Byte$.

Figure 13 shows "roofline" performance model for all GPU kernels of the GPU NICAM implementation. It is shown that output data calculations kernel, both vorticity kernels and gradient kernel give performance, close to theoretical one. This results indicate that we get maximum performance from the kernels. Performance of the input calculation kernel and divergence kernel we get is not the maximum one, but we will try to increase it by applying some additional optimization in the following work.

5. Conclusion

In this paper we have described our strategy of mapping to GPU a ultra-high resolution atmospheric model NICAM. We outlined the main steps of the mapping process and reported about results we got on TSUBAME 2.0 supercomputer.

We ported the most time-consuming modules of the initial NICAM Fortran code to a single GPU by using PGI CUDA Fortran. In this work we investigated 2-dimensional (Shallow water) case of NICAM model, and we plan to implement our GPU algorithm to the 3- dimensional case in a future work.

The results of our evaluation show a 5x speedup for a

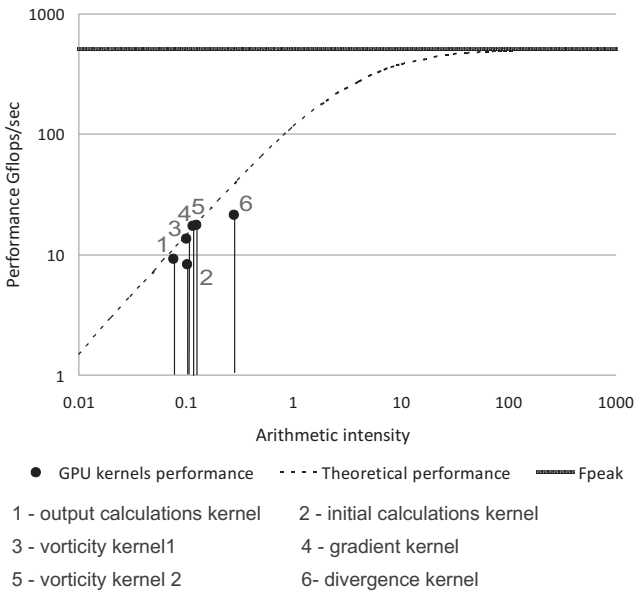


Fig. 13: Roofline graph: Visual GPU kernels performance model.

computationally intensive part of shallow water simulation model on a single GPU in comparison with a parallel CPU implementation.

In purpose to investigate performance of our GPU NICAM implementation, we created a "roofline" performance model for the GPU kernels. It was shown, that 4 of our 6 kernels give us performance close to the maximum one and, therefore, do not need any optimization. Two kernels still can be optimized to get better performance.

As our work progresses we will optimize current GPU implementation to get better performance. Then, we plan to apply our strategy to the multiple GPUs on one node, in order to reduce the amount of memory to be stored on one GPU. After that we plan to finish our MPI-CUDA implementation and run on multiple GPU nodes. We hope to see additional significant performance gains.

We also plan to investigate behavior of the NICAM code with OpenACC and the 3 compilers: PGI, CAPS/OMPP and Cray. Then it might be interesting to look at the NICAM code at the Cray XK6, which has been installed at Tokyo Tech this spring.

Acknowledgment

This work is done under the G8 ECS (Enabling Climate Simulation at Extreme Scale) project[16], focus on investigating ways to run efficiently climate simulations on future Exascale systems and get correct results.

References

[1] J. Michalakes, M. Vachharajani, "GPU Acceleration of Numerical Weather Prediction", *Parallel Processing Letters* , vol. 18 No. 4, pp. 531-548, 2008.

[2] M. Satoh, T. Matsuno, H. Tomita, H. Miura, T. Nasuno, S. Iga, "Nonhydrostatic Icosahedral Atmospheric Model (NICAM) for global cloud resolving simulations", *Journal of Computational Physics, the special issue on Predicting Weather, Climate and Extreme events*, vol. 227, pp. 3486-3514, 2007.

[3] M. W. Govett , J. Middlecoff, T. Henderson, Running the NIM next-generation weather model on GPUs", *Proc. 10th IEEE/ACM Int. Conf. Cluster, Cloud and Grid Computing (CCGrid)* , pp. 792-796, 2010.

[4] T. Shimokawabe, T. Aoki, Tomohiro Takaki, A. Yamanaka, A. Nukada, T. Endo, N. Maruyama, and S. Matsuoka, "Peta-scale Phase-Field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer", *in Proceedings of the 2011 ACM/IEEE conference on Supercomputing (SC'11)*, Nov 2011.

[5] H. Tomita, M. Satoh, "A new dynamical framework of nonhydrostatic global model using the icosahedral grid", *Fluid Dynamics Research*, vol. 34, pp. 357-400, 2004.

[6] H. Tanaka, T. BOKU, M. Satoh, "Historical Progress of the Dynamical Core of the General Circulation Model of the Atmosphere", *NAGARE: Journal of Japan Society of Fluid Mechanics*, vol. 29, N 1, pp. 26-32, 2010.

[7] R. Sadourny, A. Arakawa and Y. Mintz, "Integration of the nondivergent barotropic vorticity equation with an icosa- hedral-hexagonal grid for the sphere", *Mon. Wea. Rev.*, vol.96, pp.35-356, 1968.

[8] D.L. Williamson, J.B. Drake, J.J. Hack, R. Jakob, P.N. Swarztrauber, "A standard test set for numerical approximations to the shallow water equations in spherical geometry", *Journal of Computational Physics* , vol. 102, p. 211, 1992.

[9] H. Tomita, M. Tsugawa, M. Satoh, K. Goto, "Shallow water model on a modified icosagedral geodesic grid by using spring dynamics", *J. Comp. Phys.*, vol. 174, pp. 579-613, 2001.

[10] M. Satoh, H. Tomita, H. Miura, S. Iga, and T. Nasuno, "Development of a global cloud resolving model – a multi-scale structure of tropical convections", *J. Earth Simulator*, vol.3, pp. 11-19, 2005.

[11] E. F. Toro, emphShock-Capturing Methods for Free-Surface Shallow Flows, Wiley and Sons Ltd., 2001, 309 pages.

[12] (2012) Portland Group International CUDA Fortran Compiler. [Online]. Available: http://www.pgroup.com/resources/cudafortran.htm

[13] (2012) NVIDIA Corporation. NVIDIA CUDA Programming Guide, version 4.1.. [Online]. Available: http://www.nvidia.com/cuda

[14] (2012) The Scalasca performance toolset. [Online]. Available: http://www.scalasca.org/

[15] S. Williams, A. Waterman, D. Patterson, "Roofline: an insightful visual performance model for multicore architectures", *Commun. ACM 52*, 4, pp. 65-76, 2009.

[16] (2012) G8 ESC – Enabling Climate Simulations at Extreme Scale. [Online]. Available: http://www.inria.fr/en/news/news-from-inria/g8-enabling-climate-simulation

Numerical Solutions of Heat and Mass Transfer with the Second Kind Boundary and Initial Conditions in Capillary Porous Media Using Programmable Graphics Hardware

Hira Narang, Fan Wu and Aswad Abdul Shakur
Computer Science Department
Tuskegee University
Tuskegee, AL 36088
E-mail: narang@mytu.tuskegee.edu, wuf@mytu.tuskegee.edu and aswadat@gmail.com

***Abstract*—Nowadays, a heat and mass transfer simulation plays an important role in various engineering and industrial fields. To analyze physical behaviors of a thermal environment, we have to simulate heat and mass transfer phenomena. However to obtain numerical solutions to heat and mass transfer equations is much time-consuming. In this paper, therefore, one of acceleration techniques developed in the graphics community that exploits a graphics processing unit (GPU) is applied to the numerical solutions of heat and mass transfer equations. Implementation of the simulation on GPU makes GPU computing power available for the most time-consuming part of the simulation and calculation. The nVidia CUDA programming model provides a straightforward means of describing inherently parallel computations. This paper improves the computational performance of solving heat and mass transfer equations with the second boundary and initial conditions numerically running on GPU. We implemented simulation of heat and mass transfer using the novel CUDA platform on nVidia Quadro FX 4800 and compared its performance with an optimized CPU implementation on a high-end Intel Xeon CPU. The experimental results clearly show that GPU can perform heat and mass transfer simulation accurately and significantly accelerate the numerical calculation with the maximum observed speedups 10 times. Therefore, the GPU implementation is a promising approach to acceleration of the heat and mass transfer simulation.**

Keywords-Genereal: Numerical Solution; Heat and Mass Transfer; High Performance Computation; General Purpose Graphics Processing Unit; CUDA.

I. INTRODUCTION

During the last 4-5 decades, many scientists and engineers working in Heat and Mass Transfer processes have focused their attention to finding solutions both analytically/numerically, and experimentally. To precisely analyze physical behaviors of thermal environments, we need to simulate several heat and mass transfer phenomena such as heat conduction, convection, and radiation. A heat transfer simulation is accomplished by combining multiple computer simulations of such heat and mass transfer phenomena. With the advent of computer, initially the sequential solutions were found, and later when super-computers became available, fast solutions were obtained to above mentioned problems. However, the simulation of heat and mass transfer requires much longer execution time than the other simulations. Therefore, acceleration of the heat and mass transfer simulation is essential to realize a practical large-scale heat and mass transfer simulation.

This paper exploits the computing power of graphics processing units (GPUs) to accelerate the heat and mass transfer simulation. GPUs are cost-effective in terms of theoretical peak floating-point operation rates [1]. Therefore, comparing with expensive cluster, GPUs is a powerful co-processor on a common desktop PC that is ready to achieve a large-scale heat and mass transfer simulation at a low cost. The GPU has several key advantages over CPU architectures for highly parallel, compute intensive workloads, including higher memory bandwidth, significantly higher floating-point throughput. The GPU can be an attractive alternative to CPU clusters in high performance computing environments.

Recent announcement like CUDA [2] by nVidia proved their effort to extend both programming and memory models. CUDA (Compute Unified Device Architecture) is a new data-parallel, C-language programming API that bypasses the rendering interface and avoids the difficulties of classic GPGPU. Parallel computations are instead expressed as general-purpose, C-language kernels operating in parallel over all the points in a domain.

This paper investigates the numerical solutions to Two-point Initial-Boundary Value Problems (TIBVP) of Heat and Mass with the second boundary and initial conditions arising in capillary porous media. These problems find applications in drying processes, under-ground contaminants transport, absorption of nutrients in human bodies, transpiration cooling of space vehicles at re-entry into atmosphere, and many other science and engineering problems. Although traditional approaches of parallel-distributed processing have been applied with advantage to the solutions of some of these problems, no more seem to have explored the high performance solutions to these problems with compact multi-processing capabilities of GPU, which is multi-processors technology on a chip. With the power of this compact technology and develop relevant algorithms to find the solution of TIBVP with the second boundary and initial conditions and compare with some of the existing solutions to simple known problems. All of our experimental results show satisfactory speedups. The maximum observed speedups are about 10 times.

The rest of the paper is organized as follow: Section II introduces some previous related work; Section III describes the background on GPU and CUDA briefly; Section IV presents the mathematical model of heat and mass transfer and numerical solutions to heat and mass transfer equations;

Our experimental results are presented in Section V; Finally Section VI concludes this paper with our future direction.

II. Related Work

The simulation of heat and mass transfer has received much attention for years. And there is much work related to this field, such as modeling and dynamic simulation. Here we just refer to some recent work closely related.

Soviet Union was in the fore-front for exploring the coupled Heat and Mass Transfer in Porous media was researched as a part of chemical engineering discipline, and major advances were made at Heat and Mss Transfer Institute at Minsk, BSSR. Later England and India took the lead and made further advances in terms of analytical and numerical solutions to certain problems. Later Narang and Rajiv [4] explored the wavelet solutions and Ambethkar [5] explored the numerical solutions to some of these problems.

With the programmability of fragments on GPU, Krüger et al. [6] computed the basic linear algebra problems, and further computed the 2D wave equations and NSEs on GPU. Bolz et al. [7] rearranged the sparse matrix into textures, and utilized them multigrid method to solve the fluid problem. Similarly, Goodnight et al. [8] used the multigrid method to solve the boundary value problems on GPU. Harris [9, 10] solved the PDEs of fluid motion to get cloud animation.

GPU is also used to solve other kinds of PDEs. For example, Kim et al. [11] solved the crystal formation equations on GPU. Lefohn et al. [12] packed the level-set isosurface data into a dynamic sparse texture format, which was used to solve the PDEs. Another creative usage was to pack the information of the next active tiles into a vector message, which was used to control the vertices and texture coordinates needed to send from CPU to GPU. To learn more applications about GPU for general-purpose computations, readers can refer to [13].

III. An Overview of CUDA Architecture

The GPU that we have used in our implementations is nVidia's Quadro FX 4800, which is DirectX 10 compliant. It is one of nVidia's fastest processors that support the CUDA API and as such all implementations using this API are forward compatible with newer CUDA compliant devices. All CUDA compatible devices support 32-bit integer processing. An important consideration for GPU performance is its level of occupancy. Occupancy refers to the number of threads available for execution at any one time. It is normally desirable to have a high level of occupancy as it facilitates the hiding of memory latency.

The GPU memory architecture is shown in figure 1.

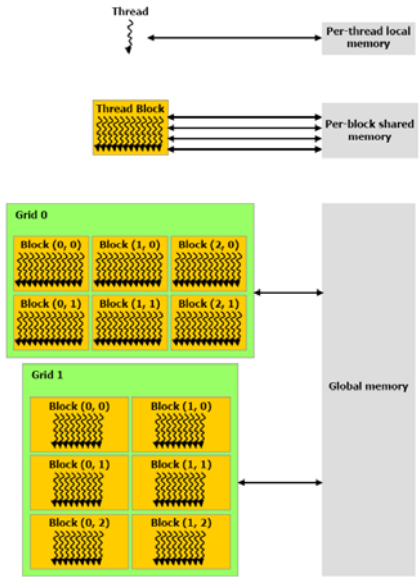


Figure 1: GPU Memory Architecture [2]

IV. Mathematical Model and Numerical Solutions of Heat and Mass Transfer

A. Mathematical Model

Consider the Heat and Mass Transfer through a porous slab with boundary conditions of the second kind. Let the x-axis be directed upward along the slab and the y-axis normal to the slab. Let u and v be the velocity components along the x- and y- axes respectively. Let us assume that the slab is accelerating with a velocity $u = Ut$ in its own plane at time $t \geq 0$. Then the heat and mass transfer equations in the Boussinesq's approximation, are:

$$\frac{\partial v_1}{\partial x_1} = 0 \tag{1}$$

$$\frac{\partial u_1}{\partial t_1} + v_1 \frac{\partial u_1}{\partial x_1} = g\beta(T_1 - T_\infty) + \frac{v\partial^2 u_1}{\partial x_1^2} - \frac{\sigma B_0^2 u_1}{\rho} \tag{2}$$

$$\frac{\partial T_1}{\partial t_1} + v_1 \frac{\partial C_1}{\partial x_1} = \frac{k\partial^2 T_1}{\partial x_1^2} \tag{3}$$

$$\frac{\partial C_1}{\partial t_1} + v_1 \frac{\partial C_1}{\partial x_1} = \frac{D'\partial^2 C_1}{\partial x_1^2} \tag{4}$$

A prescribed constant heat flux q supplied by the hot plate at the left end $X=0$ of the slab, the initial and boundary conditions of the problem are:

$$t_1 \leq 0, u_1(x_1, t_1) = 0$$
$$T_1(x_1, t_1) = T_\infty \text{ , } C_1(x_1, t_1) = C_\infty \tag{5}$$

$$t_1 > 0, u_1(0, t_1) = V_0 \tag{6}$$

$$k\frac{\partial T(0,t_1)}{\partial x_1} = -q$$
$$\frac{\partial C(0,t_1)}{\partial x_1} + k\frac{\partial T(0,t_1)}{\partial x_1} = 0$$
$$t_1 > 0 \qquad (7)$$
$$u_1(\infty,t_1) \to 0,\ T_1(\infty,t_1) \to T_\infty$$
$$C_1(\infty,t_1) \to C_\infty \text{ as } x_1 \to \infty$$

Since the slab is assumed to be porous, Equation (1) integrates to $v_1 = -v_0$ is the constant velocity. Here, μ_1 is the velocity of the fluid, T_p the temperature of the fluid near the slab, T_∞ the temperature of the fluid far away from the slab, C_p the concentration near the slab, C_∞ the concentration far away from the slab, g the acceleration due to gravity, β the coefficient of volume expansion for heat transfer, β' the coefficient of volume expansion for concentration, ν the kinematic viscosity, σ the scalar electrical conductivity, ω the frequency of oscillation, k the thermal conductivity, q is the heat flux and t_1 is the time.

From Equation (1) we observe that v_1 is independent of space co-ordinates and may be taken as constant. We define the following non-dimensional variables and parameters.

$$t = \frac{t_1 V_0^2}{4\nu}, x = \frac{V_0 x_1}{4\nu} \qquad (8)$$
$$u = \frac{u_1}{V_0}, T = \frac{T_1 - T_\infty}{T_P - T_\infty}, C = \frac{C_1 - C_\infty}{C_P - C_\infty}, P_r = \frac{\nu}{k}, S_c = \frac{\nu}{D'}$$

$$M = \frac{\sigma B_0^2 \nu}{\rho V_0^2}, G_r = \frac{\nu g\beta(T_P - T_\infty)}{V_0^3}$$
$$G_m = \frac{\nu g\beta'(C_P - C_\infty)}{V_0^3}, \omega = \frac{4\nu\omega_i}{V_0^2}$$

Now taking into account Equations (5), (6), (7), and (8), equations (2), (3) and (4) reduce to the following form:

$$\frac{\partial u}{\partial t} - 4\frac{\partial u}{\partial x} = 4\frac{\partial^2 u}{\partial x^2} + 4G_r T - 4Mu + 4G_m C \qquad (9)$$

$$\frac{\partial T}{\partial t} - 4\frac{\partial T}{\partial x} = \frac{4}{P_r}\frac{\partial^2 T}{\partial x^2} \qquad (10)$$

$$\frac{\partial C}{\partial t} - 4\frac{\partial C}{\partial x} = \frac{4}{S_c}\frac{\partial^2 T}{\partial x^2} \qquad (11)$$

with

$$t \le 0 \qquad (12)$$
$$u(x,t) = 0,\ T(x,t) = 0$$

$$t > 0 \qquad (13)$$
$$u(0,t) = 0,\ k\frac{\partial T(0,t_1)}{\partial x_1} = -q$$
$$\frac{\partial C(0,t_1)}{\partial x_1} + k\frac{\partial T(0,t_1)}{\partial x_1} = 0$$

$$t > 0 \qquad (14)$$
$$u(\infty,t) = 0,\ T(\infty,t) = 0$$
$$C(\infty,t) = 0, \text{ as } x_1 \to \infty$$

B. Numerical Solutions

Here we sought a solution by finite difference technique of implicit type namely Crank- Nicolson implicit finite difference method which is always convergent and stable. This method has been used to solve Equations (9), (10), and (11) subject to the conditions given by (12), (13) and (14). To obtain the difference equations, the region of the heat is divided into a gird or mesh of lines parallel to x and t axes. Solutions of difference equations are obtained at the intersection of these mesh lines called nodes. The values of the dependent variables T, u and C at the nodal points along the plane $x = 0$ are given by $T(0,t)$, $u(0,t)$ and $C(0,t)$ hence are known from the boundary conditions.

In the figure 2, Δx, Δt are constant mesh sizes along x and t directions respectively. We need an algorithm to find single values at next time level in terms of known values at an earlier time level. A forward difference approximation for the first order partial derivatives of u, T and C. And a central difference approximation for the second order partial derivative of u, T and C are used. On introducing finite difference approximations for:

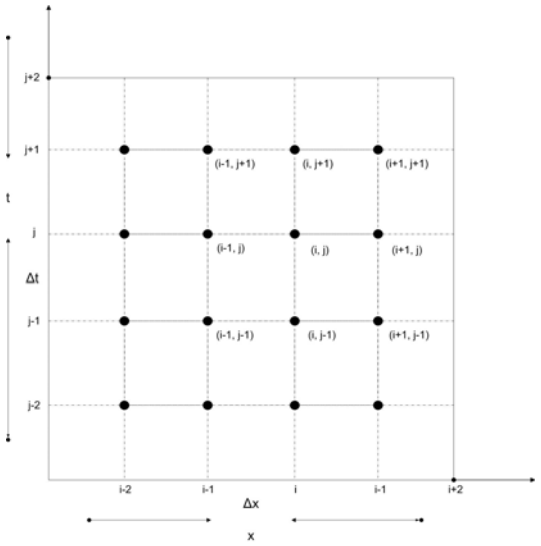


Figure 2: Finite Difference Grid

$$\left(\frac{\partial T}{\partial x}\right)_{i,j}=\frac{T_{i+1,j}-T_{i-1,j}+T_{i+1,j+1}-T_{i-1,j+1}}{4(\Delta x)}$$

$$\left(\frac{\partial C}{\partial x}\right)_{i,j}=\frac{C_{i+1,j}-C_{i-1,j}+C_{i+1,j+1}-C_{i-1,j+1}}{4(\Delta x)} \tag{15}$$

$$\left(\frac{\partial u}{\partial x}\right)_{i,j}=\frac{u_{i+1,j}-u_{i-1,j}+u_{i+1,j+1}-u_{i-1,j+1}}{4(\Delta x)}$$

$$\left(\frac{\partial T}{\partial t}\right)_{i,j}=\frac{T_{i,j+1}-T_{i,j}}{\Delta t},\left(\frac{\partial C}{\partial t}\right)_{i,j}=\frac{C_{i,j+1}-C_{i,j}}{\Delta t},\left(\frac{\partial u}{\partial t}\right)_{i,j}=\frac{u_{i,j+1}-u_{i,j}}{\Delta t}$$

$$\left(\frac{\partial^2 T}{\partial x^2}\right)_{i,j}=\frac{T_{i+1,j}+T_{i-1,j}-2T_{i,j}+T_{i+1,j+1}+T_{i-1,j+1}-2T_{i,j+1}}{2(\Delta x)^2}$$

$$\left(\frac{\partial^2 C}{\partial x^2}\right)_{i,j}=\frac{C_{i+1,j}+C_{i-1,j}-2C_{i,j}+C_{i+1,j+1}+C_{i-1,j+1}-2C_{i,j+1}}{2(\Delta x)^2}$$

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j}=\frac{u_{i+1,j}+u_{i-1,j}-2u_{i,j}+u_{i+1,j+1}+u_{i-1,j+1}-2u_{i,j+1}}{2(\Delta x)^2}$$

The finite difference approximation of Equations (9), (10) and (11) are obtained with substituting Equation (15) into Equations (9), (10) and (11) and multiplying both sides by Δt and after simplifying, we let $\frac{\Delta t}{(\Delta x)^2}=r'=1$ (method is always stable and convergent), under this condition the above equations can be written as:

$$\begin{aligned}&2u_{i,j+1}\left(\frac{1}{2}+\frac{\Delta t}{\Delta x}\right)u_{i+1,j+1}+\left(\frac{\Delta t}{\Delta x}-\frac{1}{2}\right)u_{i-1,j+1}\\&=\left(\frac{1}{2}+\frac{\Delta t}{\Delta x}\right)u_{i+1,j}+\left(\frac{1}{2}+\frac{\Delta t}{\Delta x}\right)u_{i-1,j+1}+\\&+4G_r\Delta tT_{i,j}+4G_m\Delta tC_{i,j}-4M\Delta tu_{i,j}\end{aligned} \tag{16}$$

$$\begin{aligned}&\left(1+\frac{4}{P_r}\right)T_{i,j+1}+\left(\frac{\Delta t}{\Delta x}-\frac{2}{P_r}\right)T_{i-1,j+1}-\left(\frac{\Delta t}{\Delta x}-\frac{2}{P_r}\right)T_{i+1,j+1}\\&=\left(\frac{\Delta t}{\Delta x}-\frac{2}{P_r}\right)T_{i+1,j}+\left(\frac{2}{P_r}-\frac{\Delta t}{\Delta x}\right)T_{i-1,j}+\left(1-\frac{4}{P_r}\right)T_{i,j}\end{aligned} \tag{17}$$

$$\begin{aligned}&\left(1+\frac{4}{S_c}\right)C_{i,j+1}+\left(\frac{\Delta t}{\Delta x}-\frac{2}{S_c}\right)C_{i-1,j+1}-\left(\frac{\Delta t}{\Delta x}-\frac{2}{S_c}\right)C_{i+1,j+1}\\&=\left(\frac{2}{S_c}+\frac{\Delta t}{\Delta x}\right)C_{i+1,j}+\left(\frac{2}{S_c}-\frac{\Delta t}{\Delta x}\right)C_{i-1,j}+\left(1-\frac{4}{S_c}\right)C_{i,j}\end{aligned} \tag{18}$$

V. EXPERIMENTAL RESULTS AND DISCUSSION

A. Setup and Device Configuration

The experiment was executed using the CUDA Runtime Library, Quadro FX 4800 graphics card, Intel Core 2 Duo. The programming interface used was Visual Studio.

The experiments were performed using a 64-bit Lenovo ThinkStation D20 with an Intel Xeon CPU E5520 with processor speed of 2.27 GHZ and physical RAM of 4.00GB. The Graphics Processing Unit (GPU) used was an NVIDIA Quadro FX 4800 with the following specifications:

CUDA Driver Version:	3.0
Total amount of global memory:	1.59 Gbytes
Number of multiprocessors:	24
Number of cores:	92
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	16384 bytes
Total number of registers available per block:	16384
Maximum number of threads per block:	512

Banwitdh:

Host to Device Bandwith: 3412.1 (MB/s)
Device to Host Bandwith: 3189.4 (MB/s)
Device to Device Bandwitdh: 57509.6 (MB/s)

In the experiments, we considered solving heat and mass transfer differential equations in an infinite slab with boundary conditions of the second kind using numerical methods. Our main purpose here was to obtain numerical solutions for Temperature *T*, and concentration *C* distributions across the various points in a slab as heat and mass are transferred from one end of the slab to the other. For our experiment, we compared the similarity of the CPU

and GPU results. We also compared the performance of the CPU and GPU in terms of processing times of these results.

In the experimental setup, we are given the initial temperature T_0 and concentration C_0 at point $x = 0$ on the slab. Also, there is a constant heat flux q_0 and mass flux N_0 constantly working the surface of the slab. The temperature at the other end of the slab where $x = \infty$ is assumed to be ambient temperature (assumed to be zero). Also, the concentration at the other end of the slab where $x = \infty$ is assumed to be negligible (≈ 0) and so are the heat and mass fluxes. Our initial problem was to derive the temperature T_1 and concentration C_1 associated with the heat and mass fluxes respectively. This we did by employing the finite difference technique. Hence, we obtained total initial temperature of $(T_0 + T_1)$ and total initial concentration of $(C_0 + C_1)$ at $x = 0$. These total initial conditions were then used to perform calculations.

For the purpose of implementation, we assumed a fixed length of the slab and varied the number of nodal points N to be determined in the slab. Since N is inversely proportional to the step size Δx, increasing N decreases Δx and therefore more accurate results are obtained with larger values of N. For easy implementation in Visual Studio, we employed the Forward Euler Method (FEM) for forward calculation of the temperature and concentration distributions at each nodal point in both the CPU and GPU. For a given array of size N, the nodal points are calculated iteratively until the values of temperature and concentration become stable. In this experiment, we performed the iteration for 10 different time steps. After the tenth step, the values of the temperature and concentration became stable and are recorded. We run the tests for several different values of N and Δx and the error between the GPU and CPU calculated results was increasingly smaller as N increased. Finally, our results were normalized in both the GPU and CPU.

B. *Experimental Results*

The normalized temperature and concentration distributions at various points in the slab are depicted in Table 1 and Table 2 respectively. We can immediately see that, at each point in the slab, the CPU and GPU computed results are similar. In addition, the values of temperature and concentration are highest at the point on the slab where the heat flux and mass flux are constantly applied. As we move away from this point, the values of the temperature and concentration decrease. At a point near the designated end of the slab, the values of the temperature and concentration approach zero. This is depicted clearly in the graphs of Figure 1 and Figure 2.

TABLE I. COMPARISON OF GPU AND CPU RESULTS (TEMPRETURE)

X	GPU Results	CPU Results
4.84	1.00000	1.00000
14.52	0.93147	0.93145
24.19	0.86723	0.86476
33.87	0.80299	0.80109
43.55	0.73875	0.73875
53.23	0.67451	0.67765
62.9	0.61028	0.60987
72.58	0.54604	0.54676
82.26	0.48187	0.48188
91.94	0.41756	0.41444
101.61	0.35332	0.35098
111.29	0.28908	0.28567
120.97	0.22484	0.22484
130.65	0.16067	0.15989
140.32	0.09636	0.09907
150	0.03216	0.03216

TABLE II. COMPARISON OF GPU AND CPU RESULTS (CONCENTRATION)

X	GPU Results	CPU Results
4.84	1.00000	1.00000
14.52	0.93548	0.93345
24.19	0.87097	0.87565
33.87	0.80645	0.80678
43.55	0.74194	0.74094
53.23	0.67742	0.67742
62.9	0.61298	0.63426
72.58	0.54839	0.54546
82.26	0.48387	0.48467
91.94	0.41935	0.41875
101.61	0.35484	0.35484
111.29	0.29032	0.28967
120.97	0.22581	0.22346
130.65	0.16129	0.16034
140.32	0.09677	0.09346
150	0.03226	0.03226

Furthermore, we also evaluated the performance of the GPU (NVIDIA Quadro FX 4800) in terms of solving heat and mass transfer equations by comparing its execution time to that of the CPU (Intel Xeon E5520).

For the purpose of measuring the execution time, the same functions were implemented in both the device (GPU) and the host (CPU), to initialize the temperature and concentration and to compute the numerical solutions. In this case, we measured the processing time for different values of N. The graph in Figure 3 depicts the performance of the GPU versus the CPU in terms of the processing time. We run the test for N running from 15 to 1005 with increments of 30 and generally, the GPU performed the calculations a lot faster than the CPU.

- When N was smaller than 75, the CPU performed the calculations faster than the GPU.
- When N was between 75 and 135 both CPU and GPU performed around the same speed.

- For N larger than 135 the GPU performance began to increase considerably

Figure 3 and figure 4 show some of our experimental results.

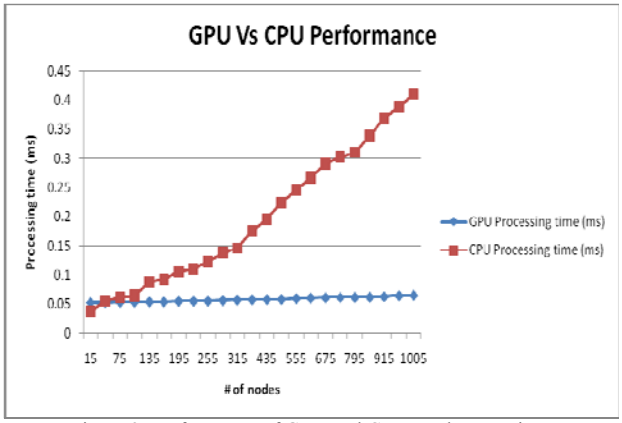


Figure 3: Performance of GPU and CPU Implementations

Finally, the accuracy of our numerical solution was dependent on the number of iterations we performed in calculating each nodal point, where more iteration mean more accurate results. In our experiment, we observed that after 9 or 10 iterations, the solution to the heat and mass equation at a given point became stable. For optimal performance, and to keep the number of iterations the same for both CPU and GPU, we used 10 iterations.

VI. Conclusion and Future Work

We have presented our numerical approximations to the solution of the heat and mass transfer equation with the second kind of boundary and initial conditions using finite difference method on GPGPUs. Our conclusion shows that finite difference method is well suited for parallel programming. We implemented numerical solutions utilizing highly parallel computations capability of GPGPU on nVidia CUDA. We have demonstrated GPU can perform significantly faster than CPU in the field of numerical solution to heat and mass transfer. Our experimental results indicate that our GPU-based implementation shows a significant performance improvement over CPU-based implementation and the maximum observed speedups are about 10 times.

There are several avenues for future work. We would like to test our algorithm on different GPUs and explore the new performance opportunities offered by newer generations of GPUs. It would also be interesting to explore more tests with large scale data set. Finally, further attempts will be made to explore more complicated problems both in terms of boundary conditions as well as geometry.

Acknowledgment

This work has been supported in part by US. NSF grant HBCU-UP.

References

[1] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A.E. Lefohn, T.J. Purcell,: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1) (2007) 80-113.

[2] NVIDIA Corporation. NVIDIA Programming Guide 2.3. Retrieved July, 2009. www.nvidia.com.

[3] A. V. Luikov. Heat and Mass Transfer in Capillary Porous Bodies, Pergamon Press, 1966.

[4] Hira Narang and Rajiv Nekkanti. Wavelet-based Solution to Time-dependent Two-point Initial Boundary Value Problems with Non-Periodic Boundary Conditions, Proceedings of the IATED International Conference Signal Processing, Pattern Recognition & Applications July 3-6 2001, Rhodes, Greece.

[5] Hira Narang and Rajiv Nekkanti. Wavelet-based Solution of Boundary Value Problems involving Hyperbolic Equations, Proceedings from the IATED International Conference Signal Processing, Pattern Recognition & Applications June 25-26, 2002

[6] Hira Narang and Rajiv Nekkanti. Wavelet-based solutions to problems involving Parabolic Equations, Proceedings of the IATED International Conference Signal Processing, Pattern Recognition & Applications 2001, Greece.

[7] Hira Narang and Rajiv Nekkanti. Wavelet-Based Solution to Elliptic Two-Point Boundary Value Problems with Non-Periodic Boundary Conditions, Proceedings from the WSEAS international conference in Signal, Speech, and Image processing Sept 25-28, 2002

[8] Hira Narang and Rajiv Nekkanti. Wavelet-Based Solution to Some Time-Dependent Two-Point Initial Boundary Value Problems with Non-Linear Non-Periodic Boundary Conditions, International Conference on Scientific computation and differential equations, SCICADE 2003, Trondheim, Norway, June 30. July 4, 2003

[9] Hira Narang and Rajiv Nekkanti. Wavelet based Solution to Time-Dependent Two Point Initial Boundary Value Problems with Non-Periodic Boundary Conditions involving High Intensity Heat and Mass Transfer in Capillary Porous Bodies, IATED International Conference proceedings, Gainesville, FL 2004.

[10] Vishwavidyalaya Ambethkar. Numerical Solutions of Heat and Mass Transfer Effects of an Unsteady MHD Free Conective Flow Past an Iffinite Vertical Plate With Constant Suction. Journal of Naval Architecture and Marine Engineering, pages 28-36, June, 2008.

[11] Jens Krüger and Rüdiger Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. ACM Transactions on Graphics (Proceedings of SIGGRAPH), pages908-916, July 2003.

[12] Jeff Bolz, Ian Farmer, Eitan Grinspun and Peter Schröoder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. ACM Transactions on Graphics (Proceedings of SIGGRAPH), pages917-924, July 2003.

[13] Nolan Goodnight, Cliff Woolley, David Luebke and Greg Humphreys. A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. In Proceeding of Graphics Hardware, pages 102-111, July 2003.

[14] Mark Harris, William Baxter, Thorsten Scheuermann and Anselmo Lastra. Simulation of Cloud Dynamics on Graphics Hardware. In Proceedings of Graphics Hardware, pages 92-101, July 2003.

[15] Mark Harris. Real-Time Cloud Simulation and Rendering. PhD thesis, 2003.

[16] Theodore Kim and Ming Lin. Visual Simulation of Ice Crystal Growth. In Proceedings of SIGGRAPH/Eurographics Symposium on Computer Amination, pages 86-97, July 2003.

[17] Aaron Lefohn, Joe Kniss, Charles Hansen and Ross Whitaker. Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware. In IEEE Visualization, pages 75-82, 2003.

[18] GPGPU website. http://www.gpgpu.org.

SESSION

COMMUNICATION SYSTEMS + NETWORKS AND INTERCONNECTION NETWORKS + PEER-TO-PEER NETWORKS + APPLICATIONS

Chair(s)

TBA

Constructing MIHCs for Arrangement Graphs $A_{n,k}$ with $n-k \geq 3$

Hsun Su[1] **and Shin-Shin Kao**[2]
[1]Department of Public Finance and Taxation,
Takming University of Science and Technology, Taipei City, Taiwan 11451, R.O.C.
E-mail: sushyun@gmail.com.
[2]Department of Applied Mathematics,
Chung-Yuan Christian University, Chung-Li, Taiwan 32023, R.O.C.
E-mail: shin2kao@gmail.com.

Abstract— *The arrangement graph $A_{n,k}$ has been used as the underlying topology for many practical multicomputers, and has been extensively studied in the past. The construction scheme of mutually independent hamiltonian cycle, abbreviated as MIHCs, on $A_{n,k}$ has not been done except for two special cases with $n-k=1$ and $n-k=2$. In this paper, we will prove that any $A_{n,k}$, where $n-k \geq 3$ and $k \geq 2$, contains $k(n-k)$ MIHCs. More specifically, let $N = \mid V(A_{n,k}) \mid$, $v(i) \in V(A_{n,k})$ for $1 \leq i \leq N$ and $\langle v(1), v(2), \cdots, v(N), v(1)\rangle$ be a hamiltonian cycle of $A_{n,k}$. We prove that $A_{n,k}$ contains $k(n-k)$ hamiltonian cycles, denoted by $C_l = \langle v(1), v_l(2), \cdots, v_l(N), v(1)\rangle$ for $1 \leq l \leq k(n-k)$, such that $v_l(i) \neq v_{l'}(i)$ for $2 \leq i \leq N$ whenever $l \neq l'$. The result is optimal since each vertex of $A_{n,k}$ has exactly $k(n-k)$ neighbors.*

Keywords: Arrangement graph, hamiltonian cycle, mutually independent.

1. Introduction

The architecture of an interconnection network is usually represented by a graph, in which vertices and edges correspond to processors and communication links, respectively. Thus, we use the terms "graph" and "network" interchangeably. Mutually independent hamiltonian cycles, abbreviated as MIHCs, and its related issues have attracted numerous studies on graphs and interconnection networks due to its many applications. (See [6], [10]–[16].) For example, in communications and in signal processing. When information has to be processed along all communication nodes, using the existence of MIHCs on an interconnection network, one can divide the information into groups and let each group be processed along different hamiltonian cycles in parallel. Obviously, it provides a more efficient way for information processing. The arrangement graph, denoted by $A_{n,k}$, is a well-studied interconnection network since it has many pleasant properties. In [2], $A_{n,k}$ was shown to be vertex symmetric and edge symmetric, $(k(n-k))$-regular with $\frac{n!}{(n-k)!}$ vertices, and contain a hamiltonian cycle. It was further shown that $A_{n,k}$ can embed multidimensional grids, hypercubes, and spanning trees all with constant dilations [4]. Readers can refer to [2]–[5], [7], [9], [17] for more results. The construction scheme of MIHCs on any arrangement graph is certainly an important task, but so far only special cases are done. Researchers have derived MIHCs on $A_{n,n-1}$ and $A_{n,n-2}$, which are called *star graphs and alternating group graphs*, respectively. (See [11] and [14].) However, a general construction scheme for MIHCs on $A_{n,k}$ with any given pair of n and k is still unknown. In this paper, we will prove that any arrangement graph $A_{n,k}$ contains $k(n-k)$ MIHCs for given integers n and k with $n-k \geq 3$ and $k \geq 2$. The result is optimal in the sense that each node of $A_{n,k}$ has exactly $k(n-k)$ neighbors and cannot have more cycles emerging from it.

2. Preliminary

For the graph definitions and notations, we follow [1]. Given a graph $G = (V, E)$, the meanings of path, cycle, hamiltonian path, hamiltonian cycle, hamiltonian graph, and hamiltonian connected graph are as usual. A path P between two vertices $v(0)$ and $v(k)$ is represented by P=$\langle v(0), v(1), ..., v(k)\rangle$, where each pair of consecutive vertices are adjacent. We also write path P=$\langle v(0), v(1), ..., v(k)\rangle$ as $\langle v(0), v(1), ..., v(i), Q, v(j), v(j+1), ..., v(k)\rangle$, where Q denotes the path $\langle v(i), v(i+1), ..., v(j)\rangle$. Let $F \subseteq V(G)$ and $G-F$ be the graph obtained by deleting v and all edges adjacent to v for all $v \in F$. If $G-F$ remains hamiltonian (resp. hamiltonian connected) for any $F \subseteq V(G)$ with $|F| \leq k$, then G is a *k-vertex-fault-tolerant hamiltonian* (resp. *k-vertex-fault-tolerant hamiltonian connected*) graph. If $G-F$ remains hamiltonian (resp. hamiltonian connected) for any $F \subseteq E(G)$ with $|F| \leq k$, then G is a *k-edge-fault-tolerant hamiltonian* (resp. *k-edge-fault-tolerant hamiltonian connected*) graph. Let $F \subseteq V(G) \cup E(G)$. If $G-F$ remains hamiltonian (resp. hamiltonian connected) for any F with $|F| \leq k$, then G is a *k-fault-tolerant hamiltonian* (resp. *k-fault-tolerant hamiltonian connected*) graph.

Two *cycles* C_1=$\langle u(1), u(2), ..., u(m), u(1)\rangle$ and C_2=$\langle v(1), v(2), ..., v(m), v(1)\rangle$ beginning at s in a graph G are *independent* if $u(1) = v(1) = s$

and $u(i) \neq v(i)$ for $2 \leq i \leq m$. Cycles beginning at s with the same length are *mutually independent* if every two different cycles are independent. A graph G is said to contain n *mutually independent hamiltonian cycles* if there exist n hamiltonian cycles in G beginning at any vertex s such that the n cycles are mutually independent.

Let $\langle n \rangle = \{1, 2, 3, ..., n\}$ and $p = p_1 p_2 ... p_k$ be a permutation of k elements in $\langle n \rangle$, where $n \geq k$. The n-dimensional arrangement graph, denoted by $A_{n,k}$, is the graph with the vertex set $V(A_{n,k})$={$p | p_i \in \langle n \rangle$ for $1 \leq i \leq k$ and $p_i \neq p_j$ if $i \neq j$}, and the edge set $E(A_{n,k})$={$(p,q) | p, q \in V(A_{n,k})$ such that $p_i \neq q_i$ and $p_j = q_j$ for all $j \neq i$ for some $i \in \langle k \rangle$}. Thus (p, q) is an edge if and only if there exists some $i \in \langle k \rangle$ such that $p_i \neq q_i$ and $p_j = q_j$ for all $j \neq i$. The k-bit label, $p_1 p_2 ... p_k$, of each vertex of $A_{n,k}$ is called the *vertex id*. See Figure 1 for an illustration.

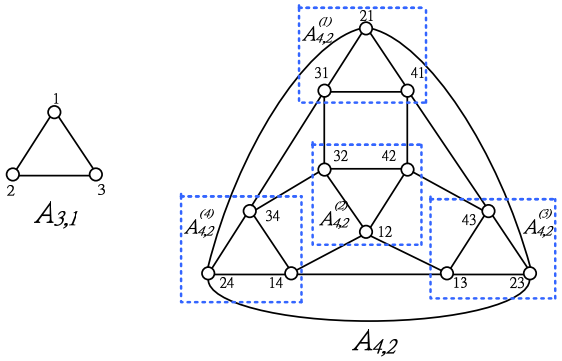


Fig. 1: The arrangement graph $A_{3,1}$ and $A_{4,2}$.

Note that each vertex of $A_{n,k}$ is represented by a k-bit sequence. In [9], it is known that $A_{n,k}$ can be decomposed into n subgraphs $A_{n-1,k-1}$ by fixing each different vertex id on one particular position among $\{1, 2..., k\}$. Without loss of generality, we partition $A_{n,k}$ into n subgraphs $A_{n-1,k-1}$, denoted by $A^{(1)}_{n,k}, A^{(2)}_{n,k}, ..., A^{(n)}_{n,k}$, by letting all vertices of $A^{(i)}_{n,k}$ with the rightmost bit being i. For example, $A_{4,2}$ consists of four copies of $A_{3,1}$. An illustration is given in Figure 1. Repeating the partitioning process, it is obvious that $A_{n,k}$ consists of $n!/(n-k+1)!$ copies of $A_{n-k+1,1}$.

Let $p = p_1 p_2 ... p_k \in V(A_{n,k})$. Note that $A^{(n)}_{n,k}$ is isomorphic to $A_{n-1,k-1}$. Moreover, it is easy to see that the k-bit id's of all vertices of $A^{(n)}_{n,k}$ can be obtained by adding one bit, the number n, to the rightmost of the $(k-1)$-bit id's of all vertices in $A_{n-1,k-1}$. That is, $\mathbf{p}n = p_1 p_2 \cdots p_{k-1} n$ is a vertex in $A^{(n)}_{n,k}$ if $\mathbf{p} = p_1 p_2 \cdots p_{k-1} \in V(A_{n-1,k-1})$.

The following proposition is an immediate consequence of the above.

Proposition 1: Let $\mathbf{u_i} \in V(A_{n-1,k-1})$ for $1 \leq \mathbf{i} \leq m$. If $\langle \mathbf{u_1}, \mathbf{u_2}, \cdots, \mathbf{u_t}, \mathbf{u_{t+1}}, \cdots, \mathbf{u_m} \rangle$ is a path of $A_{n-1,k-1}$ that passes the edge $(\mathbf{u_t}, \mathbf{u_{t+1}})$, then $\langle \mathbf{u_1}n, \mathbf{u_2}n, \cdots, \mathbf{u_t}n, \mathbf{u_{t+1}}n, \cdots, \mathbf{u_m}n \rangle$ is a path of $A^{(n)}_{n,k}$ that passes the edge $(\mathbf{u_t}n, \mathbf{u_{t+1}}n)$.

The following two propositions will be useful in our derivation.

Proposition 2: [7] Let n≥5. For $1 \leq i, j \leq n$ and $i \neq j$, the number of edges between $A^{(i)}_{n,k}$ and $A^{(j)}_{n,k}$, denoted by $|E^{i,j}_n|$, is $(n-2)!/(n-k-1)!$.

Proposition 3: [7] For positive integers n, k with $n - k \geq 2$, $A_{n,k}$ is $(k(n-k)-3)$-fault-tolerant hamiltonian connected.

3. Main Result

We will construct the $k(n-k)$ MIHCs for $A_{n,k}$ using mathematical induction. Based on the fact that $A_{n,k}$ consists of n copies of $A_{n-1,k-1}$, we must ensure that $A_{n-k+1,1}$ contains $(n-k)$ MIHCs for $n-k+1 \geq 3$. It is obvious since $A_{n-k+1,1}$ is the complete graph K_{n-k+1}. For the base case in our induction, we shall show that $A_{n,2}$ contains $2(n-2)$ MIHCs, each of which passes a common edge, in Lemma 1 and Lemma 2. Since MIHCs for $A_{n,n-1}$ and $A_{n,n-2}$ were already derived [11], [14], we concentrate on MIHCs for $A_{n,k}$ with $n - k \geq 3$ and $k \geq 2$ in Theorem 2, which is our main theorem. Figure 2 is the flowchart of the outline of our derivation.

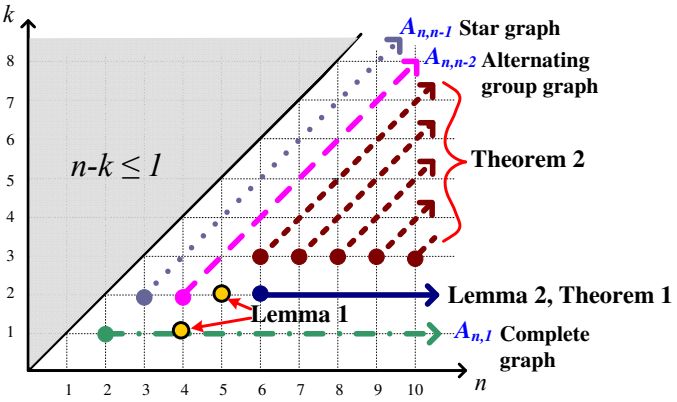


Fig. 2: Outline of the structure of lemmas and theorems in this paper.

To distinguish numbers and consecutive bits in a vertex id, we underline each bit in the vertex id should there be a possible confusion. Let $s \in V(A^{(n)}_{n,k})$ with the vertex id $s = \underline{n-k+1} \ldots \underline{n-1}\ \underline{n}$. We will show that $A_{n,k}$ contains $k(n-k)$ MIHCs. This is the maximum number of MIHCs that can be constructed on $A_{n,k}$ since any vertex in $A_{n,k}$ has exactly $k(n-k)$ neighbors. To show that $A_{n,k}$ contains $k(n-k)$ MIHCs, we construct these MIHCs beginning at any vertex u in $A_{n,k}$. Without loss of generality, let $u = s$.

Lemma 1: (1)There exist three MIHCs in $A_{4,1}$. (2)There exist six MIHCs in $A_{5,2}$, denoted by C_i, $for\ 1 \le i \le 6$, such that the edge $(21, 31)$ belongs to C_i for all $1 \le i \le 6$.

Proof: Since $A_{4,1}$ is isomorphic to the complete graph K_4, it contains three MIHCs. To prove this lemma for $A_{5,2}$, we construct six MIHCs of $A_{5,2}$ that include the edge $(21, 31)$ below. Let

$C_1 = \langle 45, 15, 14, 24, 34, 54, 51, (21, 31), 41, 42, 12, 32, 52, 53, 13, 43, 23, 25, 35, 45\rangle$;

$C_2 = \langle 45, 25, 24, 34, 54, 14, 12, 32, 42, 52, 53, 13, 23, 43, 41, 51, (21, 31), 35, 15, 45\rangle$;

$C_3 = \langle 45, 35, 34, 54, 14, 24, 23, 53, 13, 43, 41, (21, 31), 51, 52, 32, 42, 12, 15, 25, 45\rangle$;

$C_4 = \langle 45, 41, 51, (21, 31), 34, 24, 14, 54, 52, 42, 32, 12, 15, 35, 25, 23, 53, 13, 43, 45\rangle$;

$C_5 = \langle 45, 42, 52, 32, 12, 13, 53, 43, 23, 25, 15, 35, 34, 14, 24, 54, 51, (21, 31), 41, 45\rangle$;

$C_6 = \langle 45, 43, 53, 23, 13, 15, 35, 25, 24, 34, 14, 54, 51, 41, (21, 31), 32, 52, 12, 42, 45\rangle$.

Note that we add a parentheses on the required edge in C_i $for\ 1 \le i \le 6$. $\{C_i | 1 \le i \le 6\}$ are the six MIHCs in $A_{5,2}$ required by the lemma. ■

We know that $A^{(i_1)}_{n,k}$ is a subgraph of $A_{n,k}$ in which the rightmost bit of the vertex id of any vertex of $A^{(i_1)}_{n,k}$ is i_1. $A^{(i_2 i_1)}_{n,k}$ denotes a subgraph of $A^{(i_1)}_{n,k}$ in which the rightmost two bits of the vertex id of any vertex of $A^{(i_2 i_1)}_{n,k}$ is $i_2 i_1$. In fact, any vertex $i_k i_{k-1} ... i_2 i_1$ in $A_{n,k}$ can be identified as a vertex in the smallest subgraph $A^{(i_{k-1} ... i_2 i_1)}_{n,k}$. For example, 321 is a vertex in $A_{6,3}$ and it belongs to the smallest subgraph $A^{(21)}_{6,3}$. Note that $A^{(21)}_{6,3}$ is isomorphic to $A_{4,1} = K_4$.

Note that $s = \underline{n-k+1} \ldots \underline{n-1}\ \underline{n}$. In the following lemmas and theorems, s is the first (and the last) vertex of all hamiltonian cycles. For $1 \le i \le (k-1)(n-k)$, we define C^I_i as the ith hamiltonian cycle whose second vertex belongs to $A^{(n)}_{n,k}$. For $1 \le i \le (n-k)$, we define $C^O_{i+(k-1)(n-k)}$ as the $(i + (k-1)(n-k))$-th hamiltonian cycle whose second vertex is in $A^{(i)}_{n,k}$, where $i \ne n$.

In the following, we define a new symbol $[r]_n \equiv (r\ mod\ n) + 1$, where $(r \bmod n)$ denotes the remainder of n dividing r.

Lemma 2: For $n \ge 6$, there are $2(n-2)$ MIHCs in $A_{n,2}$, denoted by C_i $for\ 1 \le i \le\ 2(n-2)$, such that the edge $(21, 31)$ belongs to C_i, for all i.

Proof: Note that any vertex $u \in A_{n,2}$ is labeled by $u = u_1 u_2$. Let $n \ge 6$, we construct $2(n-2)$ MIHCs beginning at $\underline{n-1}\ \underline{n}$, denoted by $\{C_i | 1 \le i \le 2(n-2)\}$, which consists of $\{C^I_i |\ 1 \le i \le (n-2)\}$ and $\{C^O_{i+(n-2)} |\ 1 \le i \le n-2\}$.

Let $\overrightarrow{c} = (1, 2, ..., n-2)$ be a $(n-2)$-dimensional vector, $\overrightarrow{d} = (1, 2, ..., n-2, n)$ be a $(n-1)$-dimensional vector, and $c(i)$(resp. $d(i)$) denote the ith element of $\overrightarrow{c}$ (resp. $\overrightarrow{d}$). Let $P^{n-1}_i = \langle \underline{d([1+(i-2)]_{n-1})}\ \underline{n-1}, \underline{d([2+(i-2)]_{n-1})}\ \underline{n-1}, \cdots, \underline{d([(n-1)+(i-2)]_{n-1})}\ \underline{n-1}\rangle$, where P^{n-1}_i is the ith path in the subgraph $A^{(n-1)}_{n,2}$. By Proposition 2, for distinct integers i and j with $1 \le i, j \le n$, $|E^{i,j}_n| = n-2$. We choose distinct vertices $a^i, b^i \in A^{(i)}_{n,2}$, such that $\{(b^i, a^j) | 1 \le i, j \le n,\ i \ne j\} \subseteq E(A_{n,2})$ and $|\{a^1, b^1\} \cap \{21, 31\}| \le 1$. By Proposition 3, $A^{(i)}_{n,2}$, is $(2n-7)$-fault-tolerant hamiltonian connected for any $1 \le i \le n$. Thus for $1 \le i \le n-1$, there exists a hamiltonian path P^i of $A^{(i)}_{n,2}$ between a^i and b^i. Since $A^{(1)}_{n,2}$ is isomorphic to K_{n-1}, there exists a hamiltonian path P^1 between a^1 and b^1 on $A^{(1)}_{n,2}$, such that the edge $(21, 31)$ lies on P^1. For $i = 1, 2, ..., (n-2)$, we construct $(n-2)$ MIHCs as follow:

$$
\begin{aligned}
C^I_i = \langle & \underline{n-1}\ \underline{n},\ \underline{c([1+(i-2)]_{n-2})}\ \underline{n}, \\
& \underline{d([1+(i-2)]_{n-2})}\ \underline{n-1},\ P^{n-1}_i, \\
& \underline{d([(n-1)+(i-2)]_{n-1})}\ \underline{n-1}, \\
& \underline{d([(n-1)+(i-2)]_{n-1})}\ \underline{c([1+(i-2)]_{n-2})}, \\
& P^{c([1+(i-2)]_{n-2})}, b^{c([1+(i-2)]_{n-2})}, a^{c([2+(i-2)]_{n-2})}, \\
& P^{c([2+(i-2)]_{n-2})}, b^{c([2+(i-2)]_{n-2})}, \cdots, \\
& a^{c([(n-2)+(i-2)]_{n-2})}, P^{c([(n-2)+(i-2)]_{n-2})}, \\
& \underline{c([2+(i-2)]_{n-2})}\ \underline{c([(n-2)+(i-2)]_{n-2})}, \\
& \underline{c([2+(i-2)]_{n-2})}\ \underline{n}, \underline{c([3+(i-2)]_{n-2})}\ \underline{n}, \cdots, \\
& \underline{c([(n-2)+(i-2)]_{n-2})}\ \underline{n}, \underline{n-1}\ \underline{n}\rangle.
\end{aligned}
$$

Then we construct the other $(n-2)$ MIHCs as follows. By Proposition 2, for distinct integers i and j with $1 \le i, j \le n$, $|E^{i,j}_n| = n-2$. We choose $v^{c(1)} = \underline{d(n-2)}\ \underline{c(1)}$ so that the $(n+1)$th vertex $\underline{d(n-2)}\ \underline{n-1}$ of $C^O_{1+(n-2)}$ never collides with C^I_i, and distinct vertices $u^i, v^i \in A^{(i)}_{n,2}$ such that $\{(v^i, u^j) | 1 \le i, j \le n,\ i \ne j\} \subseteq E(A_{n,2})$ and $|\{u^1, v^1\} \cap \{21, 31\}| \le 1$. By Proposition 3, $A^{(i)}_{n,2}$, is $(2n-7)$-fault-tolerant hamiltonian connected for any $1 \le i \le n$. Thus there exists a hamiltonian path Q^i of $A^{(i)}_{n,2}$ between u^i and v^i for $1 \le i \le n-1$, and a hamiltonian path Q^n of $A^{(n)}_{n,2} - \{s\}$ between u^n and v^n. Since $A^{(1)}_{n,2}$ is isomorphic to K_{n-1}, there exists a hamiltonian path Q^1 between u^1 and v^1 on $A^{(1)}_{n,2}$ such that the edge $e = (21, 31)$ lies on Q^1. For $1 \le i \le (n-2)$, we define

$$C^O_{1+(n-2)} = \langle \underline{n-1}\ \underline{n}, \underline{n-1}\ \underline{c(1)}, Q^{c(1)}, \underline{d(n-2)}\ \underline{c(1)},$$
$$\underline{d(n-2)}\ \underline{n-1}, Q^{n-1}, v^{n-1}, u^{c(2)}, Q^{c(2)}, v^{c(2)},$$
$$\cdots, v^{c(n-3)}, u^n, Q^n, v^n, u^{c(n-2)}, Q^{c(n-2)},$$
$$\underline{n-1}\ \underline{c(n-2)}, \underline{n-1}\ \underline{n}\rangle.$$

For $i = 2, ..., (n-2)$, we set:

$$C^O_{i+(n-2)} = \langle \underline{n-1}\ \underline{n}, \underline{n-1}\ \underline{c([1+(i-2)]_{n-2})},$$
$$Q^{c([1+(i-2)]_{n-2})}, v^{c([1+(i-2)]_{n-2})}, \cdots,$$
$$u^{c([(n-i-1)+(i-2)]_{n-2})}, Q^{c([(n-i-1)+(i-2)]_{n-2})},$$
$$v^{c([(n-i-1)+(i-2)]_{n-2})}, u^n, Q^n, v^n, u^{n-1}, Q^{n-1},$$
$$v^{n-1}, u^{c([(n-i)+(i-2)]_{n-2})}, Q^{c([(n-i)+(i-2)]_{n-2})},$$
$$v^{c([(n-i)+(i-2)]_{n-2})}, \cdots, u^{c([(n-2)+(i-2)]_{n-2})},$$
$$Q^{c([(n-2)+(i-2)]_{n-2})},$$
$$\underline{n-1}\ \underline{c([(n-2)+(i-1)]_{n-2})}, \underline{n-1}\ \underline{n}\rangle.$$

In the construction of $C^O_{i+(n-2)}$ above, Q^j denotes a hamiltonian path between u^j and v^j in $A^{(j)}_{n,k}$, where j is a function of i. Thus even in the same $A^{(j)}_{n,k}$, we pick different u^j and v^j for $C^O_{i+(n-2)}$ as i is different. The lemma has been proved. ■

For instance, we construct the eight MIHCs of $A_{6,2}$ with the above algorithm. Let $\overrightarrow{c} = (1,2,3,4)$, $\overrightarrow{d} = (1,2,3,4,6)$, and $c(i)$(resp. $d(i)$) denote the ith element of $\overrightarrow{c}$ (resp. $\overrightarrow{d}$). Let P^5_i=$\langle d([i-1]_5)\ \underline{5}, d([i]_5)\ \underline{5}, \cdots, \underline{d([i+3]_5)}\ \underline{5}\rangle$ for $1 \le i \le 4$. For $i = 1,2,3,4$, C^I_i and C^O_{i+4} are illustrated in Figure 3 and Figure 4, respectively.

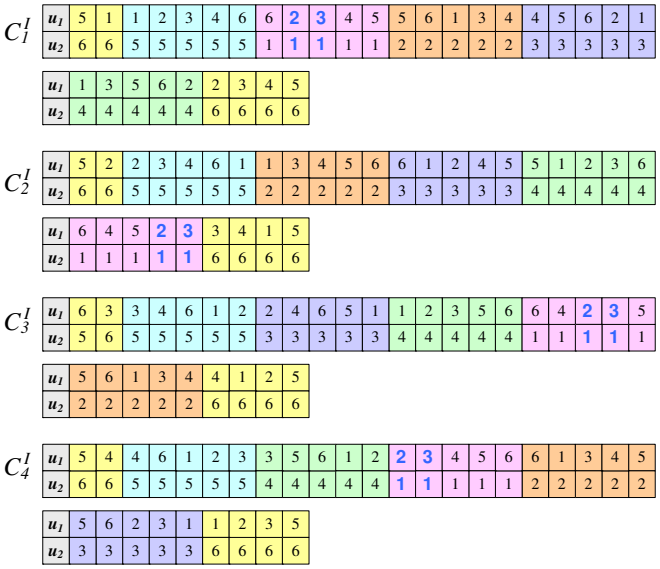


Fig. 3: An illustration of C^I_i in $A_{6,2}$.

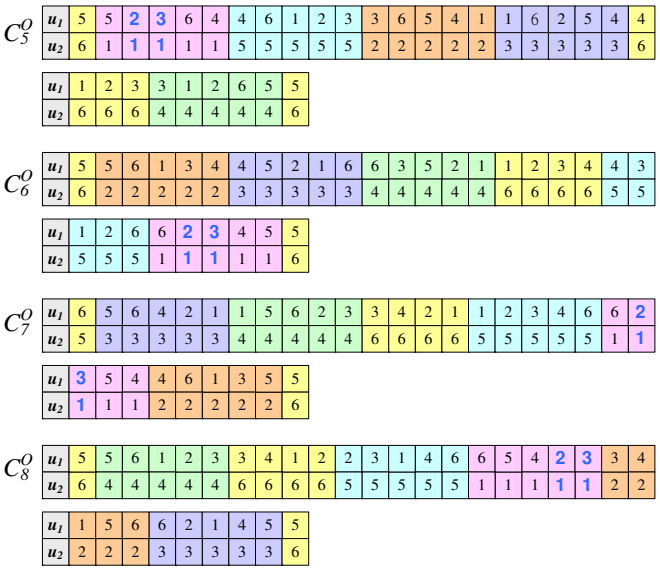


Fig. 4: An illustration of C^O_i in $A_{6,2}$.

With Lemma 1 and Lemma 2, we have the following theorem.

Theorem 1: For $n \ge 5$, there are $2(n-2)$ MIHCs in the arrangement graphs $A_{n,2}$.

Lemma 3: Let $n \ge 6$, $n-k \ge 2$ and $k \ge 3$. Given four vertices $\mathbf{u}i_2i_1$, $\mathbf{v}i_ni_1$, $\mathbf{a}i_3i_1$ and $\mathbf{b}i_3i_1$ of $A^{(i_1)}_{n,k}$, where i_2, i_3 and i_n are three distinct integers such that $\mathbf{u}i_2i_1 \in V(A^{(i_2i_1)}_{n,k})$ $\mathbf{v}i_ni_1 \in V(A^{(i_ni_1)}_{n,k})$ and ($\mathbf{a}i_3i_1$,$\mathbf{b}i_3i_1$)$\in E(A^{(i_3i_1)}_{n,k})$. Then the following two statements are true. (1) In $A^{(i_1)}_{n,k} \backslash A^{(i_2i_1)}_{n,k} \backslash A^{(i_3i_1)}_{n,k}$, there exists a hamiltonian path between $\mathbf{v}i_ni_1$ and any vertex $\mathbf{d}i_4i_1 \in V(A^{(i_4i_1)}_{n,k})$, $i_4 \notin \{i_2, i_3, i_n\}$. (2) There exists a hamiltonian path between $\mathbf{u}i_2i_1$ and $\mathbf{v}i_ni_1$ on $A^{(i_1)}_{n,k}$ passing through the assigned edge ($\mathbf{a}i_3i_1$,$\mathbf{b}i_3i_1$).

Proof: (1) By Proposition 2, for distinct integers i and j with $i, j \in \langle n\rangle \backslash \{i_1\}$, there exist $(n-3)!/(n-k-1)!$ edges between $A^{(ii_1)}_{n,k}$ and $A^{(ji_1)}_{n,k}$. By Proposition 3, $A^{(ii_1)}_{n,k}$ is hamiltonian connected for any $i \in \langle n\rangle \backslash \{i_1\}$. Let $\{i_4, i_5, ..., i_n\} = \langle n\rangle \backslash \{i_1, i_2, i_3\}$, $u^{i_4i_1}$=$\mathbf{d}i_4i_1$ and $v^{i_ni_1}$=$\mathbf{v}i_ni_1$. We choose distinct vertices $u^{i_ji_1} \in A^{(i_1)}_{n,k}$ for $5 \le j \le n$ and $v^{i_ji_1} \in A^{(i_1)}_{n,k}$ for $4 \le j \le n-1$, such that $\{(v^{i_ji_1}, u^{i_{j+1}i_1}) | 4 \le j \le n-1\} \subseteq E(A^{(i_1)}_{n,k})$. There exists a hamiltonian path $P^{i_ji_1}$ on $A^{(i_ji_1)}_{n,k}$ between $u^{i_ji_1}$ and $v^{i_ji_1}$ for any $4 \le j \le n$. We can construct a hamiltonian path on $A^{(i_1)}_{n,k} \backslash A^{(i_2i_1)}_{n,k} \backslash A^{(i_3i_1)}_{n,k}$ between $u^{i_4i_1}$ and $v^{i_ni_1}$ as $P^{i_1}_2 = \langle u^{i_4i_1}, P^{i_4i_1}, v^{i_4i_1}, u^{i_5i_1}, P^{i_5i_1}, v^{i_5i_1}, \cdots, u^{i_ni_1}, P^{i_ni_1}, v^{i_ni_1}, \rangle$. Thus there exists a hamiltonian path between $\mathbf{d}i_4i_1$ and $\mathbf{v}i_ni_1$ on $A^{(i_1)}_{n,k} \backslash A^{(i_2i_1)}_{n,k} \backslash A^{(i_3i_1)}_{n,k}$. See Figure 5 for an illustration.

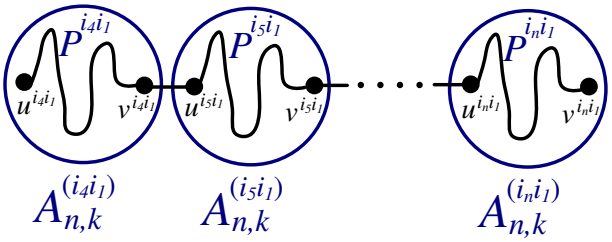


Fig. 5: An illustration of Lemma 3 (1).

(2) By Proposition 3, there exists a hamiltonian path between $\mathbf{a}i_3i_1$, $\mathbf{b}i_3i_1$ on $A^{(i_3i_1)}_{n,k}$. We set $P^{i_3i_1} = \langle \mathbf{a}i_3i_1, Q_1^{i_3i_1}, \mathbf{c}i_3i_1, \mathbf{d}i_3i_1, Q_2^{i_3i_1}, \mathbf{b}i_3i_1 \rangle$, where $(\mathbf{c}i_3i_1, \mathbf{c}i_2i_1)$, $(\mathbf{d}i_3i_1, \mathbf{d}i_4i_1) \in E(A^{(i_1)}_{n,k})$ and $\mathbf{c}i_2i_1 \neq \mathbf{u}i_2i_1$. There exists a hamiltonian path $\langle \mathbf{u}i_2i_1, P^{i_2i_1}, \mathbf{c}i_2i_1 \rangle$ on $A^{(i_2i_1)}_{n,k}$. By (1), there exists a hamiltonian path $\langle \mathbf{d}i_4i_1, P_2^{i_1}, \mathbf{v}i_ni_1 \rangle$ on $A^{(i_1)}_{n,k} \backslash A^{(i_2i_1)}_{n,k} \backslash A^{(i_3i_1)}_{n,k}$ also. So there exists a hamiltonian path $\langle \mathbf{u}i_2i_1, P^{i_2i_1}, \mathbf{c}i_2i_1, \mathbf{c}i_3i_1, (Q_1^{i_3i_1})^{-1}, \underline{\mathbf{a}i_3i_1}, \underline{\mathbf{b}i_3i_1}, (Q_2^{i_3i_1})^{-1}, \mathbf{d}i_3i_1, \mathbf{d}i_4i_1, P_2^{i_1}, \mathbf{v}i_ni_1 \rangle$ on $A^{(i_1)}_{n,k}$. See Figure 6 for an illustration. ■

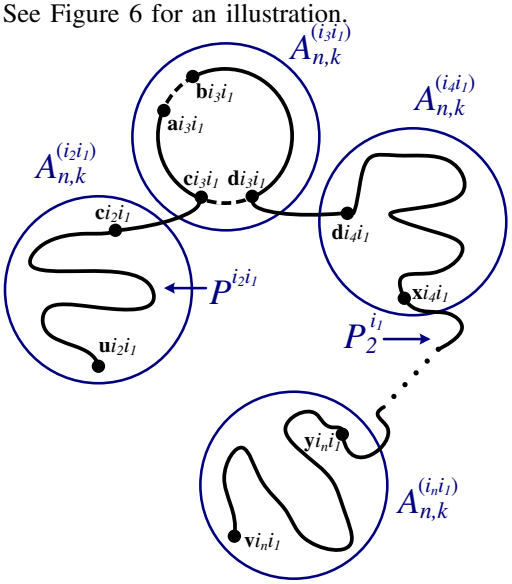


Fig. 6: An illustration of Lemma 3 (2).

Theorem 2: For $n \geq 5$, $k \geq 2$, $n - k \geq 3$, there exist $k(n-k)$ MIHCs in $A_{n,k}$, denoted by $C_1, C_2, ..., C_{k(n-k)}$, such that the edge $(\ \underline{k}\ \underline{k-1}\ \underline{k-2}\ \cdots\ \underline{1},\ \underline{k+1}\ \underline{k-1}\ \underline{k-2}\ \cdots\ \underline{1}\ )$ lies on each cycle C_i for $1 \leq i \leq k(n-k)$.

Proof: We prove this theorem by mathematical induction. By Theorem 1, the statement holds when $k = 2$. With the induction hypothesis, we assume that this theorem is true for $A_{n-1,k-1}$, where n is an integer and k is an integer with $3 \leq k \leq n-3$. It suffices to show that the statement holds for $A_{n,k}$. We first construct $(k-1)(n-k)$ MIHCs in $A_{n,k}$, denoted by $C_1^I, C_2^I, ..., C_{(k-1)(n-k)}^I$, such that the edge $(\ \underline{k}\ \underline{k-1}\ \underline{k-2}\ \cdots\ \underline{1},\ \underline{k+1}\ \underline{k-1}\ \underline{k-2}\ \cdots\ \underline{1}\ )$ is on each cycle C_i^I for $1 \leq i \leq (k-1)(n-k)$ for $n \geq 6,\ 3 \leq k \leq n-2$.

By the induction hypothesis, this theorem holds for $A_{n-1,k-1}$. There exist $(k-1)(n-k)$ MIHCs in $A^{(n)}_{n,k}$, denoted by $C_1^{'}, C_2^{'}, \cdots, C_{(k-1)(n-k)}^{'}$, such that the edge $(\ \underline{k-1}\ \underline{k-2} \cdots \underline{1}\ \ \underline{n}, \underline{k}\ \ \underline{k-2}\ \cdots \underline{1}\ \ \underline{n}\ )$ belongs to $C_i^{'}$ for all $1 \leq i \leq (k-1)(n-k)$. Assume that $a^n = \underline{k-1}\ \underline{k-2} \cdots \underline{1}\ \underline{n}$ and $b^n = \underline{k}\ \underline{k-2}\ \cdots \underline{1}\ \underline{n}$. Obviously, (a^n, b^n) appears on the $(k-1)(n-k)$ MIHCs $C_1^{'}, C_2^{'}, \cdots, C_{(k-1)(n-k)}^{'}$ at different time-steps. Let $C_i^{'} = \langle s, D^n_{i,1}, a^n, b^n, D^n_{i,2}, s \rangle$ for $1 \leq i \leq (k-1)(n-k)$, where $D^n_{i,1}$ is a path of $A^{(n)}_{n,k}$ between s and a^n, $D^n_{i,2}$ is a path of $A^{(n)}_{n,k}$ between b^n and s, and $D^n_{i,1} \bigcup D^n_{i,2}$ covers $V(A^{(n)}_{n,k})$. By Proposition 2, for distinct integers i and j with $1 \leq i, j \leq n$, $|E^{i,j}_n| = (n-2)!/(n-k-1)!$. We choose distinct vertices $a^i, b^i \in A^{(i)}_{n,k}$ for $1 \leq i \leq n$, such that $(a^n, a^{n-1}), (b^{n-1}, a^1), (b^1, a^2), \cdots, (b^{n-3}, a^{n-2}), (b^{n-2}, b^n) \in E(A_{n,k})$. By Proposition 3, $A^{(i)}_{n,k}$, which is isomorphic to $A_{n-1,k-1}$, is $((k-1)(n-k)-3)$-fault-tolerant hamiltonian connected for any $1 \leq i \leq n$, and hence is hamiltonian connected. Thus for $2 \leq i \leq n-1$, there exists a hamiltonian path P^i of $A^{(i)}_{n,k}$ between a^i and b^i. With Lemma 3, there exists a hamiltonian path P^1 between a^1 and b^1 on $A^{(1)}_{n,k}$ such that the edge $(\ \underline{k}\ \underline{k-1}\ \underline{k-2}\ \cdots\ \underline{1},\ \underline{k+1}\ \underline{k-1}\ \underline{k-2}\ \cdots\ \underline{1}\ )$ lies on P^1. For $1 \leq i \leq (k-1)(n-k)$, let

$$C_i^I = \langle s, D^n_{i,1}, a^n, a^{n-1}, P^{n-1}, b^{n-1}, a^1, P^1, b^1, a^2, P^2, b^2, ..., a^{n-3}, P^{n-3}, b^{n-3}, a^{n-2}, P^{n-2}, b^{n-2}, b^n, D^n_{i,2}, s \rangle.$$

Now we construct the rest $(n-k)$ MIHCs, which we denote by $C^O_{i+(k-1)(n-k)}$ for $1 \leq i \leq n-k$. Our method for $C^O_{i+(k-1)(n-k)}$ works successfully for all n, k except for $n \equiv 1 (\mathrm{mod}\ k)$ and $n \equiv 2 (\mathrm{mod}\ k)$. When $n \equiv 1 (\mathrm{mod}\ k)$, if we follow the same rule for $C^O_{k(n-k)}$, then the vertex next to s (the starting vertex) will be in $A^{(n-1)}_{n,k}$, which is impossible. Besides, when $n \equiv 2 (\mathrm{mod}\ k)$, it is possible that $C^O_{k(n-k)-1}$ collides with any C_i^I. Thus we will construct $C^O_{k(n-k)-1}$ and $C^O_{k(n-k)}$ with a different rule. Let $\overrightarrow{e} = (1, 2, ..., n-1, n)$. Define $\overrightarrow{c} = (1, 2, ..., n-2, n, n-1)$ as a n-dimensional vector, which is constructed by switching the $(n-1)$th element and the nth element of $\overrightarrow{e}$; $\overrightarrow{d} = (1, 2, ..., n-k-1, n-1, n-k+1, ..., n-2, n, n-k)$ as a n-dimensional vector, which is constructed by switching the $(n-k)th$ element and nth element of $\overrightarrow{c}$. Let $c(i)$ (resp. $d(i)$) denote the ith element of $\overrightarrow{c}$ (resp. $\overrightarrow{d}$). By Proposition 2, for distinct integers i and j with $1 \leq i, j \leq n$, $|E^{i,j}_n| = (n-2)!/(n-k-1)!$. We choose distinct vertices $u^i, v^i \in A^{(i)}_{n,k}$, such that $\{(v^i, u^j) | 1 \leq i, j \leq n,\ i \neq j\} \subseteq E(A_{n,k})$ and $|\{u^1, v^1\} \cap \{\underline{k}\ \underline{k-1}\ \underline{k-2}\ \cdots\ \underline{1},\ \underline{k+1}\ \underline{k-1}\ \underline{k-2}\ \cdots\ \underline{1}\}| \leq 1$. By

Proposition 3, $A^{(i)}_{n,k}$, which is isomorphic to $A_{n-1,k-1}$, is $((k-1)(n-k)-3)$-fault-tolerant hamiltonian connected for any $1 \le i \le n$, and hence is hamiltonian connected. Thus there exists a hamiltonian path Q^i of $A^{(i)}_{n,k}$ between u^i and v^i for $1 \le i \le n-1$, and a hamiltonian path Q^n of $A^{(n)}_{n,k} - \{s\}$ between u^n and v^n. With Lemma 3, there exists a hamiltonian path Q^1 between u^1 and v^1 on $A^{(1)}_{n,k}$ such that the edge $(\ \underline{k}\ \underline{k-1}\ \underline{k-2}\ \cdots\ \underline{1},\ \underline{k+1}\ \underline{k-1}\ \underline{k-2}\ \cdots\ \underline{1}\ )$ lies on Q^1. For all n with $n \ge 6$ and for $1 \le i \le (n-k-2)$, let

$$\begin{aligned} C^O_{i+(k-1)(n-k)} = \langle s, & u^{c([1+i-2]_n)}, Q^{c([1+i-2]_n)}, v^{c([1+i-2]_n)}, \\ & u^{c([2+i-2]_n)}, Q^{c([2+i-2]_n)}, v^{c([2+i-2]_n)}, \cdots, \\ & u^{c([n+i-2]_n)}, Q^{c([n+i-2]_n)}, v^{c([n+i-2]_n)}, s\rangle \\ & \text{for } i \not\equiv 1 (\text{mod } k), \end{aligned}$$

or

$$\begin{aligned} C^O_{i+(k-1)(n-k)} = \langle s, & u^{d([1+i-2]_n)}, Q^{d([1+i-2]_n)}, v^{d([1+i-2]_n)}, \\ & u^{d([2+i-2]_n)}, Q^{d([2+i-2]_n)}, v^{d([2+i-2]_n)}, \cdots, \\ & u^{d([n+i-2]_n)}, Q^{d([n+i-2]_n)}, v^{d([n+i-2]_n)}, s\rangle, \\ & \text{for } i \equiv 1 (\text{mod } k). \end{aligned}$$

For $i = n-k-1$ and $i = n-k$, we have the following three cases.

Case 1. $n \not\equiv 1(\text{mod } k)$ *and* $n \not\equiv 2(\text{mod } k)$.
If $i = n-k-1$,

$$\begin{aligned} C^O_{k(n-k)-1} = \langle s, & u^{n-k-1}, Q^{n-k-1}, v^{n-k-1}, \cdots, u^{n-3}, \\ & Q^{n-3}, v^{n-3}, u^{n-2}, Q^{n-2}, v^{n-2}, u^n, Q^n, v^n, \\ & u^{n-1}, Q^{n-1}, v^{n-1}, u^1, Q^1, v^1, \cdots, u^{n-k-2}, \\ & Q^{n-k-2}, v^{n-k-2}, s\rangle. \end{aligned}$$

If $i = n-k$,

$$\begin{aligned} C^O_{k(n-k)} = \langle s, & u^{n-k}, Q^{n-k}, v^{n-k}, \cdots, u^{n-2}, Q^{n-2}, v^{n-2}, \\ & u^n, Q^n, v^n, u^{n-1}, Q^{n-1}, v^{n-1}, u^1, Q^1, v^1, u^2, \\ & Q^2, v^2, \cdots, u^{n-k-1}, Q^{n-k-1}, v^{n-k-1}, s\rangle. \end{aligned}$$

Case 2. $n \equiv 1(\text{mod } k)$.
If $i = n-k-1$,

$$\begin{aligned} C^O_{k(n-k)-1} = \langle s, & u^{n-k-1}, Q^{n-k-1}, v^{n-k-1}, u^n, Q^n, v^n, \\ & u^{n-k+1}, Q^{n-k+1}, v^{n-k+1}, u^{n-k+2}, Q^{n-k+2}, \\ & v^{n-k+2}, \cdots, u^{n-2}, Q^{n-2}, v^{n-2}, u^{n-k}, Q^{n-k}, \\ & v^{n-k}, u^{n-1}, Q^{n-1}, v^{n-1}, u^1, Q^1, v^1, u^2, Q^2, \\ & v^2, \cdots, u^{n-k-2}, Q^{n-k-2}, v^{n-k-2}, s\rangle. \end{aligned}$$

If $i = n-k$,

$$\begin{aligned} C^O_{k(n-k)} = \langle s, & u^{n-k}, Q^{n-k}, v^{n-k}, \cdots, u^n, Q^n, v^n, u^1, Q^1, \\ & v^1, \cdots, u^{n-k-1}, Q^{n-k-1}, v^{n-k-1}, s\rangle. \end{aligned}$$

Case 3. $n \equiv 2(\text{mod } k)$.
If $i = n-k-1$,

$$\begin{aligned} C^O_{k(n-k)-1} = \langle s, & u^{n-k-1}, Q^{n-k-1}, v^{n-k-1}, u^n, Q^n, v^n, \\ & u^{n-k+1}, Q^{n-k+1}, v^{n-k+1}, Q^{n-1}, v^{n-1}, \\ & u^{n-k+2}, Q^{n-k+2}, v^{n-k+2}, \cdots, u^{n-1}, u^{n-k}, \\ & Q^{n-k}, v^{n-k}, u^1, Q^1, v^1, u^2, Q^2, v^2, \cdots, \\ & u^{n-k-2}, Q^{n-k-2}, v^{n-k-2}, s\rangle. \end{aligned}$$

If $i = n-k$,

$$\begin{aligned} C^O_{k(n-k)} = \langle s, & u^{n-k}, Q^{n-k}, v^{n-k}, \cdots, u^n, Q^n, v^n, u^1, Q^1, \\ & v^1, \cdots, u^{n-k-1}, Q^{n-k-1}, v^{n-k-1}, s\rangle. \end{aligned}$$

By Case 1, Case 2, and Case 3, the proof is finished. ■

Readers can refer to [8] for concrete examples where the desired MIHCs are constructed using the algorithm in Theorem 2.

4. Conclusion

In this work, we showed that any $A_{n,k}$ contains $k(n-k)$ mutually independent hamiltonian cycles, where and $n-k \ge 3$, $k \ge 2$. The cycles are specifically constructed using the recursive structure of $A_{n,k}$ based on the fact that $A_{n,k}$ consists of n subgraphs, $A_{n-1,k-1}$. For the same topics on special arrangement graphs with $n-k=1$, $n-k=2$, the corresponding results are presented in [11] and [14], repectively. It is natural to explore the existence of mutually independent hamiltonian cycles on a given arrangement graph when any fault occurs. That is, given a set of faulty vertices/edges on an arrangement graph, how can we construct mutually independent hamiltonian cycles that include the most non-faulty vertices in any $A_{n,k}$? The answer to the question is interesting and a thorough study for such a problem on the alternating group graphs $(A_{n,n-2})$ or the star graphs $(A_{n,n-1})$ is surely a good start.

Acknowledgement

This paper adapted the thesis of C.-D. Lin [8] by rewriting the major lemma of the main theorem and correcting the redundancies.

This work was supported in part by National Science Council of the Republic of China under Contract NSC 100-2115-M-033-005- and NSC100-2918-I-033-001. Corresponding author: S.-S. Kao.

References

[1] J. A. Bondy and U. S. R. Murty, Graph Theory with Applications, North-Holland, New York, 1980.

[2] K. Day and A. Tripathi, Arrangemnet graphs: A class of generalized star graphs, *Inform. Processing Lett.* **42**(1992) 235–241.

[3] K. Day and A. Tripathi, Embedding of cycles in arrangement graphs, *IEEE Transactions on Computers* **42**(1993) 1002–1006.

[4] K. Day and A. Tripathi, Embedding grids, hypercubes, and trees in arrangement graphs, *in: Proc. Int'l Conf. Parallel Processing*, New York, 1993, 65–72.

[5] S.-Y. Hsieh, G.-H. Chen and C.-W. Ho, Fault-Free hamiltonian cycles in faulty arrangement graphs, *IEEE Computer Society* **63**(1999) 223–238.

[6] S.-Y. Hsieh and P.-Y. Yu, Fault-free mutually independent hamiltonian cycles in hypercubes with faulty edges, *J. Combin. Optim* **13**(2007) 153–162.

[7] H.-C. Hsu, T.-K. Li, J.J.M. Tan and L.H. Hsu, Fault hamiltonicity and fault hamiltonian connectivity of the arrangement graphs, *IEEE Transactions on computers* **53**(2004) 39–53.

[8] C.-D. Lin, Construction of MIHCs on Arrangement Graphs, *Master thesis*, Department of Applied Mathematics, Chung-Yuan Christian University, Taiwan, R.O.C, 2011.

[9] C.-T. Lin, Embedding $k(n-k)$ edge-disjoint spanning trees in arrangement graphs, *J. Parallel Distrib. Comput.* **63**(2003) 1277-1287.

[10] C.-K. Lin, H.-M. Huang, L.H. Hsu and S. Bau, Mutually independent hamiltonian paths in star networks, *Networks* **46**(2005) 110–117.

[11] C.-K. Lin, H.-M. Huang, J.J.M. Tan and L.H. Hsu, Mutually independent hamiltonian cycles for the pancake graphs and the star graphs, *Discrete Math.* **309**(2009) 5474–5483.

[12] Y.-K. Shih, H.-C. Chuang, S.-S. Kao and J.J.M. Tan, Mutually independent hamiltonian cycles in dual-cubes, *J. Supercomput.* **54**(2010) 239-251.

[13] Y.-K. Shih, C.-K. Lin, D.-F. Hsu, J.J.M. Tan and L.-H. Hsu, The construction of MIH cycles in bubble-sort graphs, *Int. J. of Comput. Math.* **87**(2010) 2212-2225.

[14] H. Su, S.-Y. Chen and S.-S. Kao, Mutually independent hamiltonian cycles in Alternating group graphs, *J. Supercomput.* accepted.

[15] H. Su, J.-L. Pan and S.-S. Kao, Mutually Independent Hamiltonian Cycles in k-ary n-cubes when k is even, *Computers and Electrical Engineering* **37**(2011) 319–331.

[16] C.-M. Sun, C.-K. Lin, H.-M. Huang and L.-H. Hsu, Mutually independent Hamiltonian paths and cycles in hypercubes, *J. of Interconnect. Netw.* **7**(2006) 235–255.

[17] Y.-H. Teng, J.J.M. Tan and L.-H. Hsu, Panpositionable hamiltonicity and panconnectivity of the arrangement graphs, *Applied Math. and Comput.* **198**(2008) 414–432.

Parallel LEACH Algorithm for Wireless Sensor Networks

Yi Zhu[1], Qingmei Yao[1], Glover George[2], Shaoen Wu[2], and Chaoyang Zhang[2]
School of Computing, University of Southern Mississippi, Hattiesburg, MS, U.S.A

Abstract - *Hierarchical-based protocols are some of the most popular routing schemes used in wireless sensor networks due to their favorable, energy-saving properties. However, when it comes to large-scale wireless sensor network applications, these algorithms suffer from an increase in computational complexity and latency. In this paper, a parallel algorithm for the hierarchical-based protocol has been designed based on the Low-Energy Adaptive Clustering Hierarchy (LEACH) algorithm in order to improve the routing efficiency of wireless sensor networks. This algorithm was implemented in parallel using the C programming language and mpich2, an implementation of the Message Passing Interface (MPI) specification. The routing algorithm was evaluated for large-scale sensor networks on both a shared memory supercomputer and two Linux Beowulf clusters. The results show that the parallel implementation of the hierarchical-based protocol improves the overall performance of the routing computations, while still maintaining high energy efficiency levels compared to previous hierarchical-based methods.*

Keywords: LEACH, parallel algorithm, wireless sensor networks, routing protocol

1 Introduction

In recent years, wireless sensor networks (WSNs) have been widely used in a variety of applications ranging from home and industry to civilian and military. Technological advances in electronics have enabled the development of small, low-cost wireless sensors with signal processing and communication capabilities that are used to sense various types of information among adjacent regions. These properties have made large-scale deployments of WSNs much more possible. Large-scale WSNs consist of base stations and hundreds or thousands of small sensors with sensing, computing and wireless communications capabilities. A greater number of sensors allow for monitoring physical and environmental conditions, such as temperature, barometric pressure, presences of smoke, etc., over much larger geographical regions with both finer accuracy and an increase in resiliency through better fault-tolerance. However, sensor nodes are powered by batteries, which are constrained by their limited energy supply. Therefore, an energy-efficient approach is highly desirable in order to extend the lifetime of the sensors in the WSN.

A variety of network routing protocols for WSNs have been proposed in the literature [1], such as the direct communication protocol, the minimum-transmission-energy multi-hop routing, and the clustering-based routing protocols. In the direct communication protocol, each sensor node sends collected data directly to the base station. If the base station is very far away from the sensor nodes this kind of communication will drain the power of sensor nodes quickly, significantly reducing the system lifetime. In minimum-transmission-energy (MTE) routing, sensor nodes route data to the base station in a multi-hop routing fashion via intermediate nodes. In this case, the intermediate nodes act as routers for other nodes, in addition to sensing the environment. The drawback of MTE routing is that the nodes nearest the base station will act as routers for the majority of data sent to the base station. Thus, the power of these nodes will be drained quickly, resulting in a cascading effect on the entire network and shortening the lifetime of the WSN.

LEACH [2] is one of the first clustering based protocols for sensor networks, and it has inspired many other hierarchical routing protocols [3, 4, 5, and 7]. Even now, it is still one of the most popular hierarchical routing algorithms for WSNs. It is a self-organizing, adaptive clustering protocol that forms clusters of sensor nodes based on the received signal strength and uses local cluster heads as routers to the base station. Since clusters and routing cluster heads are adaptive (cluster heads change in each round), and only the head of the cluster transmits data to the base station, this kind of approach can balance the energy usage among all of the sensors and save significant amounts of energy across the entire WSN, increasing the lifetime of the network.

However, in the case of large-scale sensor networks which consist of hundreds or thousands of nodes, the computational and communication costs of dynamic routing protocols such as LEACH are increased significantly, and it quickly becomes a significant obstacle to energy efficiency. The less time and energy spent on routing decisions, the more time we have available for data transmissions. This necessitates high-performance, parallel algorithms and techniques applied to routing protocols. Unfortunately, little work has been done in the design and development of parallel routing schemes for sensor networks. In this work, we develop a novel, parallel hierarchical-based routing protocol based on the LEACH algorithm for sensor networks to achieve high routing efficiency. This approach has been developed using the C programming language and mpich2. The routing algorithm was evaluated for large-scale sensor

networks on both a shared memory supercomputer and two Linux Beowulf clusters. The performance of our approach has been analysed and evaluated based on simulations of a radio propagation model and a circuit energy cost model. The simulations show that the parallel hierarchical-based routing protocol, as compared to previous-work in this field, can significantly improve the performance of routing without a loss of energy-efficient properties.

In this paper, we first give an introduction to the serial LEACH algorithm in Section 2. Parallel hierarchical-based routing design is given in Section 3. Section 4 mainly describes the implementation environment. Section 5 describes the simulation model. Performance Evaluations and Analysis are done in Section 6. Finally, we conclude our work in Section 7.

2 Serial Hierarchical-Based Routing Protocol

Hierarchical-based routing protocols cluster the nodes so that cluster heads with relatively high energy can collect data from other low energy nodes and transmit data to the base station. In this way, hierarchical-based routing is able to achieve the goals of energy savings and lifetime extension. LEACH is one of the first and the most popular hierarchical routing approaches for sensor networks. We choose it as our basic serial program. To better describe the parallel algorithm, we first briefly introduce the LEACH algorithm and then elaborate on the decomposition and task scheduling involved with designing parallel hierarchical-based routing.

The following description is the main idea behind the LEACH algorithm. Cluster heads change randomly over time in order to balance the energy consumption of the sensor nodes. The optimal percentage of nodes to be chosen as cluster heads is 5%. Each node will randomly choose a number between 0.0 and 1.0. If the number is less than some predefined threshold, the node is chosen to be a cluster head for the current round. The equation is as follows:

$$R(n) = \begin{cases} \frac{p}{1-p*(r \bmod \frac{1}{p})} & \text{if } n \in F \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where p is the desired percentage of cluster heads, r is the current round, and F is the set of nodes which have not been chosen as cluster heads in the last 20 rounds. Each elected cluster head then broadcasts a message to the rest of the nodes in the network to notify them that they are the new cluster heads. After receiving the message, all non-cluster head nodes will chose the cluster with the closest cluster head and join the cluster. When the setup phase is stable, the sensor nodes can begin sensing and forwarding data to the cluster heads. The cluster heads receive data from each of the nodes in the cluster, aggregate this data, and subsequently send it to the base station. After a certain period of time, the network will go back into the setup phase, performing another round of choosing cluster heads, and then returning to the stable phase in which data transmission can occur. This process continues, until all of the nodes in the network have exhausted their energy supply.

There are three steps in one round of the serial LEACH algorithm.

STEP 1: Select heads

The head node selections are based on two requirements. If the node meets the two requirements, it will be selected as a head node.

The two requirements are:

1. Generate a random number uniformly from 0.0 to 1.0. If the random number is less than $R(n)$ as in formula (1), where p is the desired percentage of heads then we designate it as a head node. We set p as 0.05, and r is the current round.

2. The node has not been selected as a head node in the past 20 rounds.

STEP 2: Clustering

Check the distance from each node to each head node (based on received signal strength). Then choose the nearest head node as its cluster head. The number of clusters equals the number of head nodes.

STEP 3: Transmission

Every node in the cluster transmits data to its respective cluster head. Each cluster head then gathers and compresses all of the information and sends it to the base station.

3 Parallel Hierarchical-Based Routing

In the routing of large-scale sensor networks, which have hundreds or thousands of sensor nodes, the serial LEACH algorithm will have to go through every sensor node to select the heads of clusters and solve a large clustering problem. This can have a severe impact on the performance of a time sensitive routing procedure such as LEACH. Therefore, it is desirable to decompose the head selection and clustering tasks onto multiple processors, and have each processor perform a sub-task to improve the performance.

3.1 Data Decomposition

The initial data of every sensor node consists of four parts:

1. The location_coordinate.

2. The flag_head that indicates whether it is the head of the cluster.

3. The flag_head_log which indicates whether it is chosen to be a head in the last 20 rounds.

4. The flag_cluster indicating which cluster it belongs to.

These data are gathered in an array, where each element has four attributes as described above. The input data array is partitioned into equal-size sub-arrays, which are sent to each processor. Figure 1 shows the data decomposition scheme.

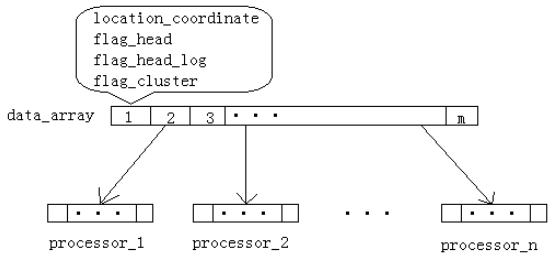


Figure 1. Data decomposition scheme

3.2 Task Dependency

The two main tasks in each round are head selection and clustering. Head selection is an independent task but clustering highly depends on the results of head selection, which can be illustrated by the analysis of the requirements and results of the two tasks. The head selection procedure requires the data array, but not necessarily the entire data array, because it does not depend on other sensors whether a sensor will be selected to be a head or not. In other words, it does not depend on other elements of the data array. Results of head selection are stored in the flag_head attribute of the data array. The clustering procedure requires all the results of head selection, since it has to check the distance from each sensor node to each head in order to determine which head is the closest one. The results of clustering are stored in flag_ cluster in the data array. The two main tasks are summarized as follows:

Head Selection Task:

Requirements: Partial data array, not necessarily the entire data array
Results: The decisions about which sensors are heads
Task dependency: Selection is an independent task

Clustering Task:

Requirements: All the results of head selection task
Results: The decisions about which head or cluster it belongs to.
Task dependency: Clustering highly depends on the results of head selection.

3.3 Parallel LEACH Algorithm Design

STEP 1: Data Decomposition

Assume there are *n* sensor nodes, data array (*data_array (i), i = 1, 2… n*) and p processors. The data array with length n is partitioned into *p* sub_arrays with equal length *n/p (sub_array(j,i), j = 1,2,...,p i = 1,2,..., n/p)*. The sub-arrays are scattered to each of the *p* processors.

STEP 2: Head Selection

Each processor selects head nodes from its own local sub-array according to the same selection process as defined in the serial implementation of LEACH, and save the result in the flag_head attribute of the local sub-array. We must also record the history of whether it has been chosen as a head node in the flag_head_log.

For each processor
If every element in flag_head_log is 0
Generate a random number r from [0.0, 1.0] uniformly.
R：= 0.05/ (1-0.05(r mod 20))
If R<threshold
flag_head = 1
flag_head_log for current round = 1
Else
flag_head = 0

STEP 3: Gather and Update data_array

Gather all the local sub-arrays from each processor into the data_array. Then scatter the data_array to every processor. At this step, each parallel task holds the information of all head nodes.

STEP 4: Clustering

Each processor completes the clustering task on their own local sub-arrays based on the information of all head nodes. The results are stored in flag_cluster.

For each node in sub-arrays
Calculate the distance between itself and every head node
Choose the head node with minimum distance
This node belongs to the cluster of this head
flag_cluster = cluster_index

STEP 5: Gather All Results and Go to the Next Round

Gather all the local sub-arrays from each processor into the global data array. Then go to STEP 1 to repeat the above steps.

4 Parallel Implementation Environment

LEACH was first simulated in serial by Wendi, et al using Matlab. Our parallel approach was implemented using the C programming language and the mpich2 implementation of MPI. The Simulations were conducted on two platforms available to students of the University of Southern Mississippi (USM). One is the Albacore cluster which is the primary HPC cluster in the School of Computing at USM [6]. The cluster consists of 256 processor cores, 300GB of RAM and 1Gbit Ethernet interconnects. Albacore is a hybrid, distributed-shared memory cluster, consisting primarily of Intel Xeon 56xx processors, Intel Xeon 55xx processors. The second platform available to our research group was Sequoia, an HPC Linux cluster at the University of Mississippi's Center for

Super Computing Research (MCSR) [8]. Sequoia is a hybrid, distributed-shared memory system, consisting of. 84 compute nodes, 22 Altix XE 310, 24 Altix 320 and 38 Rackable computing nodes. The overall memory of Sequoia is 2.1 TB.

5 Simulation Model

In this section, we describe the simulation model used for performance analysis of our parallel implementation. In order to maintain a similar comparison, the same simulation model is used for both the serial and parallel implementations. The environment configuration and energy cost model for the sensor networks are described.

5.1 Environment Configuration

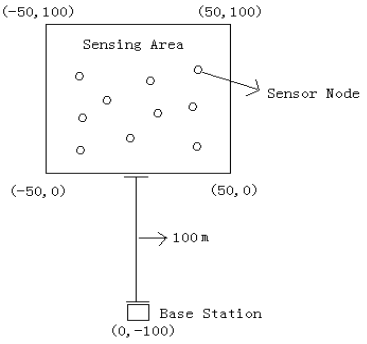


Figure 2 The coordinates of sensor areas and base station

The sensors are deployed uniformly in a 100-by-100 meter area. Figure 2 shows an example of a sensor area, along with its corresponding base station, along with coordinates. The base station is allocated 100 meters away from the sensing area, which is specifically at the location of (0, -100) in Figure2.

5.2 Energy Cost Model

The initial energy of each sensor node is 50J. In each round, every sensor node sends 2000 bits data. The energy cost of transmission consists of two parts:

1. The energy cost of transmitter and receiver circuit

$$E_{elec} = 50nJ / bit \qquad (2)$$

Every bit the circuit transmits and receives will cost 50nJ energy.

2. The energy loss of radio propagation

$$\varepsilon_{amp} = \frac{100\,pJ}{bit} / m^2 \qquad (3)$$

The energy loss of radio propagation is inversely proportional to the square of the distance it travels. In addition to this loss, the total energy consumed is also proportional to the number of bits transmitted.

The formulas of energy cost at both transmitter and receiver side are given below.

$$E_{Tx}(k,d) = E_{elec} * k + \varepsilon_{amp} * k * d^2 \qquad (4)$$

$$E_{Rx}(k) = E_{elec} * k \qquad (5)$$

E_{Tx} and E_{Rx} denote the energy cost at transmitter and receiver respectively. k is the number of bits transmitted.

6 Performance Evaluations and Analysis

6.1 System life time

The hierarchical-based routing protocol's main advantage is its high energy-efficiency, which supports a much longer system lifetime than other routing protocols. Therefore, the parallel hierarchical-based routing solution should reflect the same energy saving advantages as the serial algorithm. There are two important attributes that indicate the system stability and lifetime. One attribute is the number of head nodes in the sensor network at each round, as the energy cost of the network is highly dependent on the number and length, both distance and bit-wise, of the data transmissions. Since only the head nodes of the system do the majority of transmissions, it is important to keep track of how many of these are active at any time in the system. Although the nodes have data compression capability, the head nodes still have to transmit a relatively large amount of data, which is usually a long distance, energy consuming transmission. The other attribute is the number of sensor nodes at each round which have not exhausted their energy supply. The definition of system lifetime is the period of time that there is at least one live sensor node.

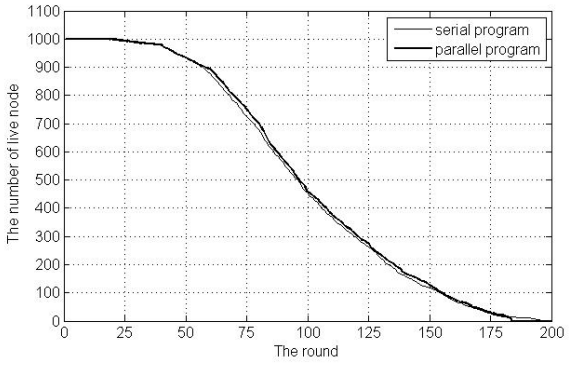


Figure 3 Number of live nodes in every round of both serial and parallel programs

Both the parallel and serial programs use the same header selection strategy, which involved a random procedure. Therefore the number of head nodes in each round can be slightly different, but the trend should be similar. In Figure 3, the number of live nodes decreases over time as the overall number of nodes decreases due to drained batteries. The variations in the number of live nodes between the serial and parallel plots are due

to the random selection of head nodes. However, the general trends of the two plots match well. In addition, the initial total energies of both the sensor networks are the same so that the lifetimes of system are the same, i.e. 200.

6.2 Performance Analysis

To evaluate the performance of our parallel hierarchical-based routing algorithm, the simulation method presented in Section 5 is employed. The parallel program was executed on Albacore and Sequoia separately using four different initial node counts (320, 640, 960 and 1280). The location and environmental conditions, including the initial energies of the sensors, were consistent across all simulation runs. The serial and parallel run times, denoted as T_p and T_s, are recorded from the experiment and shown in Tables 1 and 2, as well as in Figure 4 and 5. Derived variables used to analyse the performance are the speedup and efficiency [9, 10]. The speedup is defined as $S = T_s / T_p$; the efficiency is defined as S / p, where p is the number of processors. The reasons why the speedup and efficiency are chosen are that, for speedup, it delineates how much performance gain is achieved via parallel design over serial design; and for efficiency, it describes how much time spent on the computation, not on idle time or communication, for each processor.

Table 1: The execution times with different number of processors on Sequoia (*p* is the number of processors and column *N* is the data size)

	P=1	P=2	P=4	P=8	P=12
N=320	2.672	1.514	1.140	1.204	0.809
N=640	10.161	5.476	3.188	2.907	2.839
N=960	22.252	11.764	6.417	5.748	5.063
N=1280	39.331	20.451	11.008	6.662	6.706
N=1600	60.958	31.655	17.005	9.501	7.742

Table 2: The execution times with different number of processors on Albacore (p is the number of processors and column N is the data size)

	p=1	p=2	p=4	p=8	p=12
N=320	2.727	1.706	1.193	1.005	1.001
N=640	9.249	5.245	3.094	2.408	2.210
N=960	19.974	10.808	6.186	3.880	3.830
N=1280	34.720	18.479	10.151	6.637	6.113
N=1600	53.644	28.396	15.409	9.811	8.668

Table 3: Speedup with different number of processors on Sequoia (*p* is the number of processors and column *N* is the data size)

	p=1	p=2	p=4	p=8	p=12
N=320	1.000	1.765	2.343	2.218	3.300
N=640	1.000	1.855	3.186	3.494	3.578
N=960	1.000	1.891	3.467	3.870	4.394
N=1280	1.000	1.923	3.572	5.902	5.864
N=1600	1.000	1.925	3.584	6.415	7.873

Table 4: Speedup with different number of processors on Albacore (*p* is the number of processors and column *N* is the data size)

	p=1	p=2	p=4	p=8	p=12
N=320	1.000	1.597	2.284	2.713	2.723
N=640	1.000	1.763	2.989	3.841	4.184
N=960	1.000	1.848	3.228	5.146	5.214
N=1280	1.000	1.878	3.420	5.230	5.679
N=1600	1.000	1.889	3.481	5.467	6.188

Table 5: Efficiency with different number of processors on Sequoia (*p* is the number of processors and column *N* is the data size)

	p=1	p=2	p=4	p=8	p=12
N=320	1.000	0.882	0.585	0.277	0.275
N=640	1.000	0.927	0.796	0.436	0.298
N=960	1.000	0.945	0.866	0.483	0.366
N=1280	1.000	0.961	0.893	0.737	0.488
N=1600	1.000	0.962	0.896	0.801	0.656

Table 6: Efficiency of programs with different number of processors on Albacore (*p* is the number of processors and column *N* is the data size)

	p=1	p=2	p=4	p=8	p=12
N=320	1.000	1.597	2.284	2.713	2.723
N=640	1.000	1.763	2.989	3.841	4.184
N=960	1.000	1.848	3.228	5.146	5.214
N=1280	1.000	1.878	3.420	5.230	5.679
N=1600	1.000	1.889	3.481	5.467	6.188

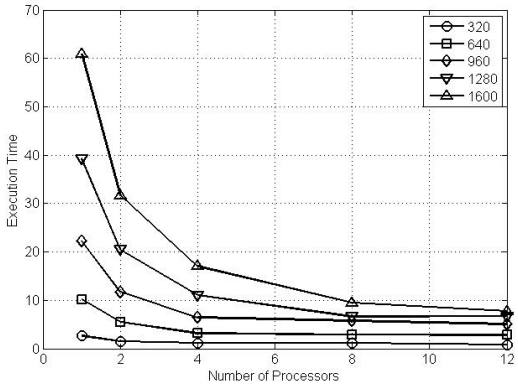


Figure 4 The execution time with different number of processors on Sequoia

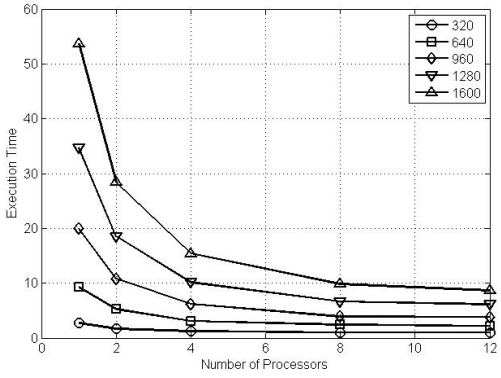


Figure 5 The execution time with different number of processors on Albacore

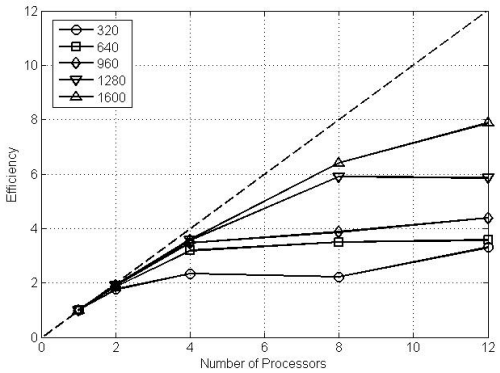


Figure 6 The speedup with different number of processors on Sequoia

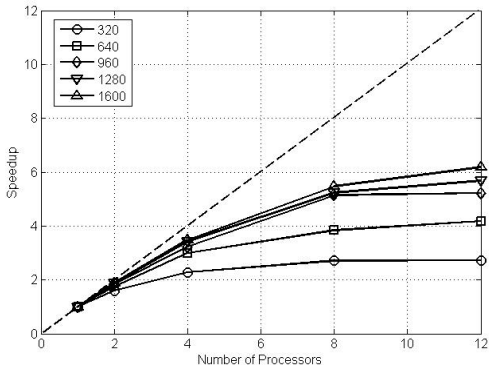


Figure 7 The speedup with different number of processors on Albacore

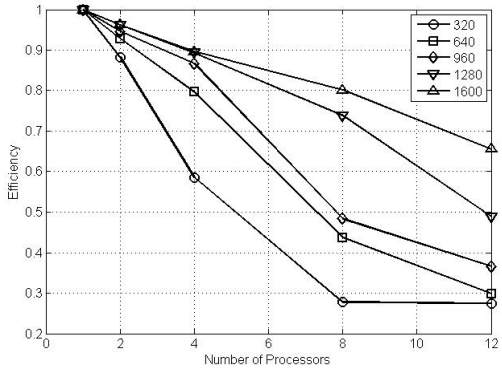


Figure 8. The efficiency with different number of processors Sequoia

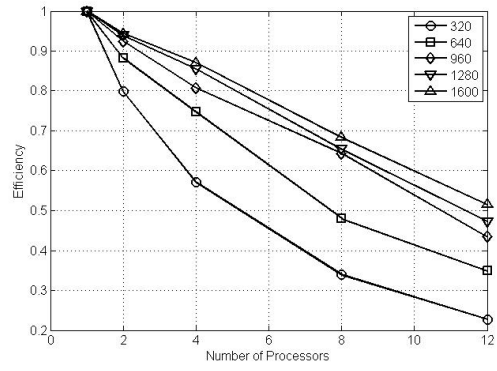


Figure 9 The efficiency with different number of processors Albacore

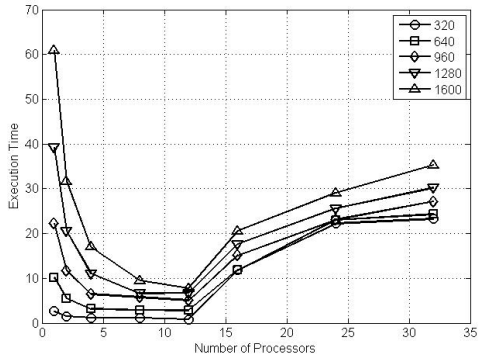


Figure 10 The execution with different number of processors on Sequoia (up to 32 processors)

From Figures 4 and 5, we can see that the execution time drops very fast for large problem size, for instance, 1280 and 1600. Nevertheless, when the number of processors is above 8, the decreasing of the execution time is less significant than that below 8. This is consistent with the speedup. Figures 6 and 7 show that, for different numbers of nodes, the speedup increases as the number of processors increases. However, the slope decreases, when the number of processor is over 8. The reason is that the communication time increases as the number of processors increases. Figures 8 and 9 show that the parallel LEACH algorithm have almost the same efficiency [11] on both Albacore and Sequoia systems. This is because these two systems have nearly identical specifications in memory and processors. Figure 10 shows that the execution time increases when the number of processors is greater than 12. This is due to the fact that both Sequoia and Albacore are hybrid, distributed-shared memory systems. Each node in the cluster consists of a single motherboard with dual-six core Intel Xeon 56xx class processors. Therefore, when executing this algorithm, the communication costs are negligible within a single node (where memory is shared), but cross-node communication costs, which occurs over 1Gbit Ethernet, is much higher than that within a single node.

7 Conclusion

The parallel hierarchical-based routing algorithm was designed and implemented in MPI based on the serial LEACH algorithm to achieve high performance routing. The task is partitioned and scheduled among all the available processors to improve the performance. The simulation results show that the high-performance computing techniques and parallel implementation can achieve a significant speedup while maintaining the energy-efficient properties of hierarchical-based routing protocols.

8 Acknowledgement

Thanks to the School of Computing at USM and the Mississippi Center for Supercomputing Research (MCSR) for providing state-of the art, high performance computing facilities and excellent services for supporting this research.

9 References

[1] K. Akkaya and M. Younis, "A survey on routing protocols for wireless sensor networks," Ad Hoc Networks, vol. 3, no. 3, pp. 325--349, May 2005.

[2] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energy-effcient communication protocol for wireless microsensor networks," Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS '00), Hawaii, January 2000.

[3] V. Loscri, G. Morabito, and S. Marano, "A Two-Level Hierarchy for Low-Energy Adaptive Clustering Hierarchy," DEIS Department, University of Calabria.

[4] S. Lindsey and C. S. Raghavendra, "PEGASIS: power efficient gathering in sensor information systems," in the Proceedings of the IEEE Aerospace Conference, Big Sky, Montana, March 2002.

[5] S. Lindsey, C. S. Raghavendra, and K. Sivalingam, "Data gathering in sensor networks using the energy*Delay metric," in the Proceedings of the IPDPS Workshop on Issues in Wireless Networks and Mobile Computing, San Francisco, CA, April 2001.

[6] http://albacore.st.usm.edu:8080/platform.

[7] Y. Jia, L. Zhao, and B. Ma, "A hierarchical clustering based routing protocol for wireless sensor networks supporting multiple data aggregation qualities," International Journal of Sensor Networks. vol. 4, no. 1/2, pp. 79-91, 2008

[8] Mississippi Center for Supercomputing Research, Available: http://www.mcsr.olemiss.edu.

[9] A. Grama, A. Gupta, G. Karypis, and V. Kumar, Introduction to Parallel Computing, Addison-Wesley, 2003.

[10] C. P. Kruskal, L. Rudolph, and M. Snir. "A complexity theory of efficient parallel algorithms," Technical Report RC13572, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1988.

[11] V. Kumar and A. Gupta. "Analysing scalability of parallel algorithms and architectures," Journal of Parallel and Distributed Computing, 22(3):379–391, 1994.

A Fault-tolerant Routing Algorithm using Directed Probabilities in Hypercube Networks

Manabu Myojin Keiichi Kaneko
Department of Computer and Information Sciences
Graduate School of Engineering
Tokyo University of Agriculture and Technology
Koganei-shi, Tokyo, JAPAN

Abstract ***In this paper, we propose a new fault-tolerant routing algorithm for hypercube networks based on directed probabilities. The directed probabilities are obtained by improving the probabilities proposed by Al-Sadi et al. The probability represents ability of routing to any node at a specific distance. Each node selects one of its neighbor nodes to send a message by taking the probabilities into consideration. We also conducted a computer experiment to verify the effectiveness of our algorithms.***

Keywords: multicomputer, interconnection network, parallel processing, fault-tolerant routing, hypercube, performance evaluation

1 Introduction

Recently, large-scaled computation is required in many fields, and interests in researches on parallel processing are increasing. Among them, studies on massively parallel systems where many processors are connected to execute computation are eagerly conducted. With a progressive increase in the number of processors in the system, probability of existence of faulty processors increases. Hence, it is necessary for massively parallel systems to establish a fault-free routing path. Such path construction is formulated to be a fault-tolerant routing problem in graph theory mapping processors and links between them to nodes and edges, respectively.

In this study, we focused on hypercubes [8, 9], which have simple and recursive structure and low diameter, and provide interconnection networks suitable for massively parallel systems. Figure 1 shows an example of a 4-dimensional hypercube Q_4. In an interconnection network, if each node can collect information of all the faulty nodes, shortest-path routing between any pair of nodes is possible. However, this method asks each node to store information of all the faulty nodes, and it requires too much memory space. In addition, collection of such information takes too much time complexity for communication.

To solve the problem, there are many routing approaches based on restricted global information where each node stores some restricted information of faulty nodes and executes quasi-optimal routing [1, 2, 3, 4, 5, 6, 7, 10, 11]. The approach by Al-Sadi et al. [1] is one such approach. In their approach, each node calculates probability of unreachability by minimal paths to destination nodes with each Hamming distance, exchanges these probabilities with its neighbors, and routes based on these probabilities. Although, their approach attains high reachability in a hypercube with faulty nodes, we believe that there remains some room for improvement. Therefore, in this study, we introduce a notion of directed routing probability, and try to improve the approach by Al-Sadi et al.

The rest of this paper is structured as follows. First, we survey related works in Section

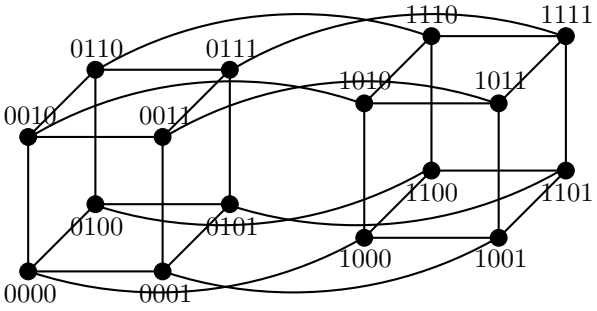


Figure 1: An example of 4-dimensional hypercube Q_4.

2. Next, requisite terminology and notations are defined in Section 3. Then, in Section 4, we introduce the algorithm proposed by Al-Sadi et al. We proposed a fault-tolerant routing algorithm that is obtained by improving the algorithm by Al-Sadi et al. in Section 5. The algorithm is evaluated by a computer experiment in Section 6. Finally, we give a conclusion and future works in Section 7.

2 Related Works

For these two decades, there are many attempts in research for fault-tolerant routing in hypercube networks. Chiu and Wu have proposed an efficient fault-tolerant routing algorithm by recursively classifying non-faulty nodes into safe, ordinary unsafe, and strongly unsafe nodes depending on the classification of neighbor nodes [5]. Chiu and Chen have improved the algorithm by introducing the routing capabilities that are obtained by classifying the safety nodes with respect to the Hamming distance to the destination nodes [6]. Wu has also proposed a similar fault-tolerant routing algorithm independently by introducing the safety vectors [10]. Moreover, Kaneko and Ito have proposed a fault-tolerant routing algorithm based on classification of ordinary and strongly unsafe nodes with respect to the Hamming distance as well as an efficient method to obtain classification of them [7].

All of the above attempts are based on information if a message is surely routed to the destination node or not. On the other hand, Al-Sadi et al. have proposed a fault-tolerant routing algorithm that is based on probabilities that a message is sent from the source node to the destination node with a path of length of Hamming distance between them [1, 2]. In the algorithm, each non-faulty node exchanges information at most $O(n^2)$ times with its neighbor nodes to calculate the probabilities with respect to the Hamming distances to destinations. However, there are several cases where the algorithm fails to find a fault-free path even if there is a such path.

3 Preliminaries

In this section, we define a hypercube network and introduce requisite notations.

Definition 1 An n-dimensional hypercube Q_n is an undirected graph, and Q_n consists of 2^n nodes. Each node $\boldsymbol{a}$ in Q_n is an n-bit sequence $(a_1, a_2, \ldots, a_n)$ where $a_i \in \{0, 1\}$ $(1 \leq i \leq n)$, and a_i is called the bit of i-th dimension. For two nodes $\boldsymbol{a}$ and $\boldsymbol{b}$ in Q_n, there is an edge $(\boldsymbol{a}, \boldsymbol{b})$ between them if and only if the Hamming distance between them $H(\boldsymbol{a}, \boldsymbol{b})$ is equal to 1. □

In general, a path in a graph is represented by an alternate sequence of nodes and edges $\boldsymbol{a}_1$, $(\boldsymbol{a}_1, \boldsymbol{a}_2)$, $\boldsymbol{a}_2$, …, $\boldsymbol{a}_{k-1}$, $(\boldsymbol{a}_{k-1}, \boldsymbol{a}_k)$, $\boldsymbol{a}_k$. The length of the path P is the number of edges included in the path, and it is denoted by $L(P)$. If Q_n is fault-free, the length of the shortest path between $\boldsymbol{a}$ and $\boldsymbol{b}$ is equal to $H(\boldsymbol{a}, \boldsymbol{b})$.

Definition 2 For a node $\boldsymbol{a}$ in Q_n, a set of nodes $N(\boldsymbol{a})$ defined by

$$N(\boldsymbol{a}) = \{\boldsymbol{n} \mid H(\boldsymbol{a}, \boldsymbol{n}) = 1\}.$$

is called a set of neighbor nodes of $\boldsymbol{a}$. □

The neighbor node of $\boldsymbol{a}$ that is obtained by changing the i-th bit $(1 \leq i \leq n)$ is denoted by $\boldsymbol{a}^{(i)}$ in the rest of paper.

In a hypercube Q_n with a set of faulty nodes F, for a source node $\boldsymbol{s}$ and a destination node $\boldsymbol{d}$

that are both non-faulty, a fault-tolerant routing algorithm finds a fault-free path between $\boldsymbol{s}$ and $\boldsymbol{d}$.

Definition 3 For two nodes $\boldsymbol{a}$ and $\boldsymbol{b}$ in Q_n, the set of preferred neighbor nodes of $\boldsymbol{a}$ for $\boldsymbol{b}$ is denoted by $N_0(\boldsymbol{a},\boldsymbol{b})$, and is defined by $N_0(\boldsymbol{a},\boldsymbol{b}) = \{\boldsymbol{n} \mid \boldsymbol{n} \in N(\boldsymbol{a}), H(\boldsymbol{n},\boldsymbol{b}) = H(\boldsymbol{a},\boldsymbol{b}) - 1\}$. In addition, the set of spare neighbor nodes of $\boldsymbol{a}$ for $\boldsymbol{b}$ is denoted by $N_1(\boldsymbol{a},\boldsymbol{b})$, and is defined by $N_1(\boldsymbol{a},\boldsymbol{b}) = \{\boldsymbol{n} \mid \boldsymbol{n} \in N(\boldsymbol{a}), H(\boldsymbol{n},\boldsymbol{b}) = H(\boldsymbol{a},\boldsymbol{b}) + 1\}$. □

Note that, in Q_n, the number of nodes that are apart from a node $\boldsymbol{a}$ by Hamming distance h is equal to ${}_nC_h$. Note also that, for two nodes $\boldsymbol{a}$ and $\boldsymbol{b}$ in Q_n, if $H(\boldsymbol{a},\boldsymbol{b}) = h$, then $|N_0(\boldsymbol{a},\boldsymbol{b})| = h$ holds.

4 Algorithm by Al-Sadi et al.

In this section, we give the idea of the algorithm by Al-Sadi et al. In their algorithm, for an arbitrary non faulty node $\boldsymbol{a}$ in an n-dimensional hypercube Q_n with a faulty node set F, an estimate value of the probability that there is not any fault-free minimal path from $\boldsymbol{a}$ to an arbitrary node $\boldsymbol{b}$ such that $H(\boldsymbol{a},\boldsymbol{b}) = h$ is calculated, and routing is executed based on the estimated values.

For simplicity of explanation, we use an estimate value of the probability that there is some fault-free path instead of the probability that there is not any fault-free path in our paper. In addition, we assume that F represents a faulty node set in Q_n in the rest of the paper.

$P_h(\boldsymbol{a})$ represents an estimate value of probability of existence of a fault-free minimal path from a fault-free node $\boldsymbol{a}$ to an arbitrary node at Hamming distance h from $\boldsymbol{a}$. $P_h(\boldsymbol{a})$ is defined recursively:

$$P_h(\boldsymbol{a}) = \begin{cases} |N(\boldsymbol{a}) \setminus F|/n & (h = 1) \\ 1 - \prod_{i=1}^{n}(1 - R_h(\boldsymbol{a}^{(i)})) & \\ \qquad (2 \le h \le n, \boldsymbol{a} \notin F) & \end{cases}$$

$$R_h(\boldsymbol{b}) = \begin{cases} 0 & (\boldsymbol{b} \in F) \\ hP_{h-1}(\boldsymbol{b})/n & (\boldsymbol{b} \notin F) \end{cases}$$

The estimate values of probabilities can be calculated by the algorithm shown in Figure 2.

```
function EVP(a, h, F)
begin
  if h = 1 then P_h(a) := |N(a) \ F| / n
  else begin
    P := 1;
    for i := 1 to n do begin
      if a^(i) ∈ F then R_h(a^(i)) := 0
      else R_h(a^(i)) := h × P_{h-1}(a^(i)) / n;
      P := P × (1 - R_h(a^(i)))
    end;
    P_h(a) := 1 - P
  end;
  return P_h(a)
end
```

Figure 2: Function to calculate estimate values of probabilities

Their routing algorithm `route` based on the estimate values of probabilities is shown in Figure 3. When a node $\boldsymbol{a}$ has to forward a message to its destination $\boldsymbol{d}$, the algorithm is used.

The algorithm first checks whether the current node $\boldsymbol{a}$ is the destination node $\boldsymbol{d}$ itself or not. If $\boldsymbol{a} = \boldsymbol{d}$, the message is delivered to $\boldsymbol{d}$ immediately. Otherwise, if $\boldsymbol{d}$ is a neighbor node of $\boldsymbol{a}$ ($\boldsymbol{d} \in N(\boldsymbol{a})$), the message is also delivered to $\boldsymbol{d}$ immediately. If $H(\boldsymbol{a},\boldsymbol{d}) \ge 2$, the algorithm tries to find the preferred neighbor node of $\boldsymbol{a}$ that has the largest estimate value of probability. If the value is positive, then the message is forwarded to the node. Otherwise, if it is 0, then the spare neighbor nodes are checked to find one with the largest estimate value. If the value is not 0, the message is sent to the node while if it is 0, the delivery fails.

Their algorithm attains high reachability in Q_n with faulty nodes. However, there remains some room for improvement. That is, the value $P_{h-1}(\boldsymbol{a})$ may has an effect on $P_h(\boldsymbol{a}^{(i)})$. Hence, $\boldsymbol{a}$ may send a message to its neighbor $\boldsymbol{a}^{(i)}$ even if $P_h(\boldsymbol{a}^{(i)}) = 0$ without $P_{h-1}(\boldsymbol{a})$. Figure 4

```
function route(a, d, F)
begin
  h := H(a, d);
  if h = 0 then begin
    deliver the message to a; exit
  end;
  if h = 1 then begin
    send the message to d; exit
  end;
  i0 := argmax_i{P_{h-1}(a^(i)) | a^(i) ∈ N0(a, d)};
  if a^(i0) ∉ F then begin
    deliver the message to a^(i0); exit
  end;
  i1 := argmax_i{P_{h+1}(a^(i)) | a^(i) ∈ N1(a, d)};
  if a^(i1) ∉ F then begin
    deliver the message to a^(i1); exit
  end;
  error('message delivery failed')
end
```

Figure 3: Routing algorithm based on estimate values of probabilities

shows an example of this case. In the figure, $P_h(\boldsymbol{a}^{(i)})$ may be positive if $P_{h-1}(\boldsymbol{a})$ is also positive though there is no way out from $\boldsymbol{a}^{(i)}$. Therefore, in the next section, we introduce a notion of directed routing probability, and try to improve the algorithm by Al-Sadi et al.

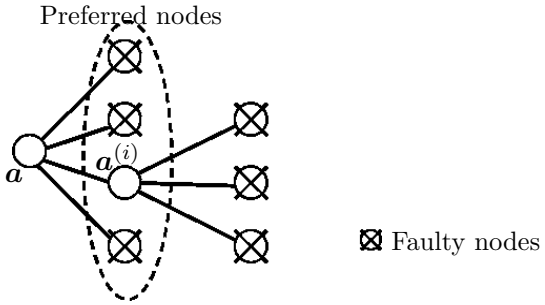


Figure 4: An example of an unreachable case

5 Proposed Algorithm

To address the problem mentioned at the end of the previous section, we have excluded an effect by $P_{h-1}(\boldsymbol{a})$ from calculation of $P_h(\boldsymbol{a}^{(k)})$. The new probability is called directed probability, and we calculate estimate values of directed probabilities $\vec{P}_h^k(\boldsymbol{a})$ where k represents the direction to which the estimate value is transmitted.

$$\vec{P}_h^k(\boldsymbol{a}) = \begin{cases} |(N(\boldsymbol{a}) \setminus \{\boldsymbol{a}^{(k)}\}) \setminus F|/(n-1) & (h=1) \\ 1 - \prod_{\substack{i=1 \\ i \neq k}}^{n} (1 - \vec{R}_h^i(\boldsymbol{a}^{(i)})) & (2 \leq h \leq n, \boldsymbol{a} \notin F) \end{cases}$$

$$\vec{R}_h^i(\boldsymbol{b}) = \begin{cases} 0 & (\boldsymbol{b} \in F) \\ h\vec{P}_{h-1}^i(\boldsymbol{b})/(n-1) & (\boldsymbol{b} \notin F) \end{cases}$$

The estimate values of directed probabilities can be calculated by the algorithm shown in Figure 5.

```
function EVDP(a, h, k, F)
begin
  if h = 1 then
    P⃗_h^k(a) := |(N(a)\{a^(k)})\F| / (n - 1)
  else begin
    P⃗ := 1;
    for i := 1 to n do begin
      if i = k then continue;
      if a^(i) ∈ F then R⃗_h^i(a^(i)) := 0
      else R⃗_h^i(a^(i)) := h × P⃗_{h-1}^i(a^(i)) / (n - 1);
      P := P × (1 - R⃗_h^i(a^(i)))
    end;
    P⃗_h^k(a) := 1 - P
  end;
  return P⃗_h^k(a)
end
```

Figure 5: Function to calculate estimate values of directed probabilities

Our routing algorithm `route2` based on the estimate values of directed probabilities is shown in Figure 6. It is similar to the algorithm by Al-Sadi et al., and it is used when a node $\boldsymbol{a}$ has to forward a message to its destination $\boldsymbol{d}$.

The time complexity to calculate estimate values of directed probabilities $\vec{P}_1^k(\boldsymbol{a})$, $\vec{P}_2^k(\boldsymbol{a})$,

```
function route2(a, d, F)
begin
  h := H(a, d);
  if h = 0 then begin
    deliver the message to a; exit
  end;
  if h = 1 then begin
    send the message to d; exit
  end;
  i_0 := argmax_i{P^i_{h-1}(a^(i)) | a^(i) ∈ N_0(a, d)};
  if a^(i_0) ∉ F then begin
    deliver the message to a^(i_0); exit
  end;
  i_1 := argmax_i{P^i_{h+1}(a^(i)) | a^(i) ∈ N_1(a, d)};
  if a^(i_1) ∉ F then begin
    deliver the message to a^(i_1); exit
  end;
  error('message delivery failed')
end
```

Figure 6: Routing algorithm based on estimate values of directed probabilities

..., $\vec{P}^k_n(\boldsymbol{a})$ $(k = 1, 2, \ldots, n)$ is $O(n^3)$, which is larger than the time complexity of the approach by Al-Sadi et al. The number of times that a node exchanges these values with its neighbor nodes is $O(n^2)$. It is same as that by Al-Sadi et al.

6 Evaluation

To evaluate performance of our algorithm, we carried out an computer experiment to compare it with the algorithm by Al-Sadi et al. The experiment is conducted based on the following procedure:

1. In Q_n, for the ratio of faulty nodes $\alpha = 0, 0.1, \ldots, 1.0$, repeat Steps 2 to 4 for 10,000 times.

2. Set $\lfloor \alpha 2^n \rfloor$ faulty nodes in Q_n.

3. Select two nodes $\boldsymbol{s}$ and $\boldsymbol{d}$ randomly such that there exists a fault-free path between them.

4. Apply the algorithms `route` and `route2`, and check if the message is delivered to the destination or not.

Figure 7 shows the result of the experiment with Q_{10}. According to the figure, we can see that our algorithm is superior to the algorithm by Al-Sadi et al. for $\alpha = 0.6, 0.7, 0.8$ and 0.9. The reason that the ratios of successful routings increased for $\alpha = 0.9$ is that the ratio of faulty nodes is so high that the pairs of $\boldsymbol{s}$ and $\boldsymbol{d}$ could be found with short Hamming distance only.

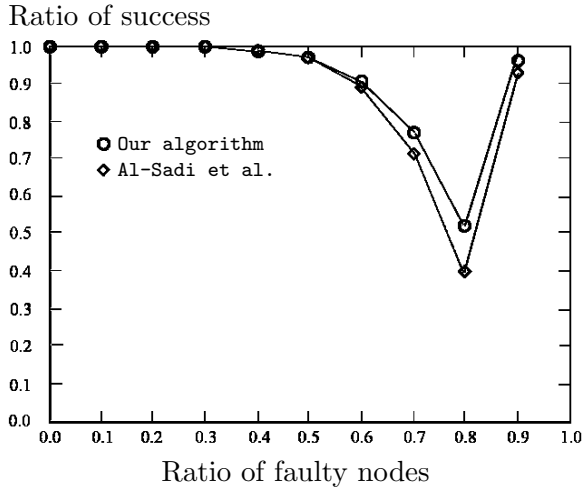


Figure 7: Ratio of successful routings by algorithms `route` and `route2` in Q_{10}

7 Conclusion

In this paper, we have proposed a new fault-tolerant routing algorithm for hypercube networks based on directed probabilities, which are obtained by improving the probabilities proposed by Al-Sadi et al. The time complexity to calculate all of the estimate values of directed probabilities is $O(n^3)$. The number of times for each node to exchange these values with its neighbors is $O(n^2)$. We also conducted a computer experiment to verify the effectiveness of our algorithm, and we showed that our algorithm is superior to that by Al-Sadi et al. when the ratio of faulty nodes α is relatively high ($\alpha = 0.6, 0.7, 0.8$, and 0.9).

Future works include reduction of the time complexity of the algorithm. Applying similar approach to other topologies is also included in future works.

Acknowledgement

This study was partly supported by a Grant-in-Aid for Scientific Research (C) of the Japan Society for the Promotion of Science under Grant No. 22500041.

References

[1] J. Al-Sadi, K. Day, and M. Ould-Khaoua. Probability-based fault-tolerant routing in hypercube. *The Computer Journal*, Vol. 44, No. 5, May 2001.

[2] J. Al-Sadi, K. Day, and M. Ould-Khaoua. Fault-tolerant routing in hypercubes using probability vectors. *Parallel Computing*, Vol. 27, pp. 1381–1399, July 2001.

[3] M. S. Chen and K. G. Shin. Depth-first search approach for fault-tolerant routing in hypercube multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, pp. 152–159, April, 1990.

[4] M. S. Chen and K. G. Shin. Adaptive fault-tolerant routing in hypercube multicomputers. *IEEE Transactions on Computers*, Vol. 39, pp. 1406–1416, April, 1990.

[5] G.-M. Chiu and S.-P. Wu. A fault-tolerant routing strategy in hypercube multicomputers. *IEEE Transactions on Computers*, Vol. 45, No. 2, pp. 143–155, Feb. 1996.

[6] G.-M. Chiu and K.-S. Chen: Use of routing capability for fault-tolerant routing in hypercube multicomputers. *IEEE Transactions on Computers*, Vol. 46, No. 8, pp. 953–958, Aug. 1997.

[7] K. Kaneko and H. Ito. Fault-tolerant routing algorithms for hypercube interconnection networks. *IEICE Transactions on Information and Systems*, Vol. E84-D, No. 1, pp. 121–128, Jan. 2001.

[8] Y. Saad and M. H. Schultz: Topological properties of hypercubes. *IEEE Transactions on Computers*, Vol. 37, No. 7, pp. 867-872, July 1988.

[9] C. L. Seitz. The cosmic cube. *Communications of ACM*, Vol. 28, No. 7, pp. 22–33, July 1985.

[10] J. Wu: Adaptive fault-tolerant routing in cube-based multicomputers using safety vectors. *IEEE Transactions on Parallel and Distributed Systems*, 9(4):322–334, Apr. 1998.

[11] Dong Xiang. Fault-tolerant routing in hypercube multicomputers using local safety information. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12, No. 9, September, 2001.

Stabilizing Information Dissemination in Wireless Sensor Networks

Sain Saginbekov and Arshad Jhumka
Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK

Abstract— *One important component of network reprogramming is code dissemination, when the new program code is distributed to the relevant nodes. Very few information dissemination protocols, if any, tolerate transient data faults, i.e., faults that corrupt the memory state of nodes. We address this limitation by proposing a novel protocol, called* Repair*, that transforms any fault-intolerant information dissemination protocol into a stabilizing protocol where, eventually, all nodes obtain the updated code. We conduct simulations to show the performance of Repair, and we integrate Repair with Varuna, and our results show that Repair induces low overhead on Varuna, and causes all nodes to receive the new code. Our main contribution is the first corrector protocol that stabilizes information dissemination in the presence of transient faults.*

Keywords: Information dissemination; Transient faults; Stabilization; Wireless sensor networks; Composition

1. Introduction

Wireless sensor networks (WSNs) have enabled the deployment of several novel classes of applications, such as monitoring and tracking. However, to be useful, they need to operate unattended for long periods of time. However, this operational mode places several requirements on the network applications, but mainly that the applications are able to adapt to changing conditions. Given that WSNs are often deployed in hostile environments, human intervention is impossible. Thus, over-the-air reprogramming becomes a fundamental activity.

A network reprogramming protocol consists of several specific components: (i) a component that decides whether a complete code needs to be sent or only an update, (ii) information dissemination component, and (iii) reliability components [1]. The reliability component is to ensure that nodes receive all the parts of the code update that may be lost due to collisions. In this paper, we focus on the information dissemination aspect of network reprogramming.

Several information dissemination protocols have been proposed as part of network reprogramming protocols [2], [3], [4], [5]. However, to the best of our knowledge, none of them is *fault-tolerant*, i.e., none of them tolerate transient faults that corrupt the memory state of the nodes, which are known to occur in WSNs [6], [7], [8]. Such transient faults are also called soft errors. Given that several information dissemination protocols work by advertising the metadata of the new code, e.g., [2], [3], [9], any corruption of the metadata can, in the worst case, lead to the network nodes having stale code. Thus, it is important to have make these information dissemination protocols fault-tolerant. One trivial way of addressing this problem is to have the base station to periodically initiate the dissemination of the new code. However, this is a very expensive process, not suitable for WSNs.

In this work, instead of proposing a new fault-tolerant information dissemination protocol for WSNs, we propose a *corrector* protocol, called *Repair*, that, when integrated with a fault-intolerant information dissemination protocol, transforms the protocol into a stabilizing [10] fault-tolerant dissemination protocol, i.e, a protocol that guarantees that, eventually, all nodes will download the correct code. The novelty of our approach is that Repair enables the deployment of several stabilizing fault-tolerant information dissemination protocols. The corrector protocol, Repair, is triggered only when an erroneous state exists in the network, thus no overhead is incurred when no transient failures occur.

Contributions: In this context, we make the following contributions:

- We present a corrector protocol called *Repair* that, when integrated with any fault-intolerant information dissemination protocol, generates a corresponding stabilising fault-tolerant information dissemination protocol, and we prove its correctness.
- We run simulation experiments on *Repair* using TOSSIM [11], and show the performance of Repair, especially its locality property.
- We present a case study where we compose *Repair* with an existing dissemination algorithm, namely Varuna [3]. We show that our protocol (Repair) induces very little overhead over the Varuna in presence of transient faults. Further, Varuna, when executed in presence of even a single transient fault resulted in all the nodes downloading the wrong code. In contrast, when running Varuna with *Repair*, all the nodes eventually obtained the new code.

The paper is structured as follows: In Section 2, we present an overview of related work. In Section3, we present the system and fault models assumed in the paper. In Section 4, we present a corrector algorithm that stabilises the information dissemination of code updates. We present the simulation setup to evaluate the performance of the proposed algorithm in Section 5. The simulation results are presented in Section 6. In Section 7, we present a case

study where we integrate the algorithm with an existing information dissemination protocol to show the viability of our approach. We conclude the paper in Section 8, and present some avenues for future work.

2. Related Work

Information Dissemination There exist several dissemination protocols which update nodes'codes to new ones. While some of the protocols deliver complete binary image of the code like [12], [13], [9], [4], some deliver only the difference between the new code and the old code [14], [15]. Also there exist protocols which deliver tasks [16], network parameters [17], and queries [18].

In XNP[12], the base station broadcasts the code image to the nodes which are in its coverage range. The nodes outside of the range cannot receive the code image. The protocol proposed in MOAP[13] is a multihop dissemination protocol that can deliver code images to nodes that are several hops away from the base station. Each node forwards the code image further after receiving the complete code image. Deluge [9], allows large data transmission by fragmenting data into fixed-size pages. It also supports pipelining page transmission to make dissemination faster. Unlike MOAP, nodes in Deluge should not wait for complete code image before forwarding it. Authors of MNP [4] proposes another protocol like Deluge, which fragments the code image and uses pipelining mechanism. However, unlike Deluge, MNP selects the sender of the code such that there is only one sender at a time in a neighbourhood. Sender selection reduces collision and hidden terminal problem. Also, in MNP, some of the nodes can go to sleep mode to save energy whenever there is no data to receive and transmit.

Because of the feature of a wireless sensor network, such as transient link failures and node mobility, not all nodes update their code to the newest one during dissemination phase. The Trickle algorithm [2] addresses this problem by using a "polite gossip"policy. In Trickle, every node broadcasts advertisement messages, a metadata that includes version number of the code, at most once per period given between [τ/2, τ]. If a node hears more than k identical metadata before it transmits, it suppresses its broadcast and doubles the value of τ up to τ_h, which is upper bound for τ. If it hears different metadata, τ becomes τ_l, which is lower bound for τ. Varuna [3] is another protocol which supports code update maintenance. This protocol saves energy in steady phase, the phase where no dissemination is being done. Unlike Trickle, where there is a linear increase of energy consumption, energy consumption in Varuna is constant in steady phase. To achieve constant energy consumption in steady phase, nodes in Varuna send advertisement messages only when there is a change in neighbourhood topology or metadata since its last advertisement transmission. In Varuna, a node detects a fault when its version is less than that of the sender. These protocols do not consider transient memory faults, which may lead to a protocol to work incorrectly. For example, a node with new code may download old code if such a fault occurs. To the best of our knowledge, the work presented in this paper is the first to address fault-tolerant information dissemination. **Fault Tolerance** In [19], it has been shown that a class of components, known as correctors, is sufficient to design non-masking fault tolerance. Thus, stabilisation, which is a special type of non-masking fault tolerance, is achieved by adding corrector mechanisms to a program, thereby transforming the program into a non-masking fault-tolerant one. Correctors are components that enforce a given predicate on program executions, whenever the predicate has been violated. The area of self-stabilization is mature, and several stabilizing algorithms exist [10].

3. Models: System and Faults

Graphs and networks: We define a wireless sensor node, or node, as a computing device equipped with a wireless interface and associated with a unique identifier. A node can communicate with a set of other nodes that lie at a certain distance from it. Generally, communication in wireless networks is typically modelled with a circular communication range centred on the node. However, we do not assume that all nodes have the same communication range and we do not assume that the range is circular. In this model, a node is thought as able to exchange data with all devices within its communication range.

A wireless sensor network is a collection of wireless sensor nodes and is modelled as an directed graph $G = (V, E)$ where V is a set of $N = |V|$ wireless sensor nodes and E is a set of edges or links, each link being a pair of distinct nodes. A node $n \in V$ is said to be a 1-hop neighbour of a node $m \in V$ iff $(m, n) \in E$, i.e., m can send a message to n. Observe that communication need not be symmetric, i.e., if m can send a message to n, n may not be able to do so. We denote by M, the set of m's 1-hop neighbours (or neighbours, for short). We say that two nodes m and n can collide at node p if $(p \in M) \wedge (p \in N)$ [1].

Faults: A fault model stipulates the way programs may fail. We consider *transient data faults* that corrupt the state of the program by artificially corrupting the values held by variables. These faults are also known as soft errors.

Definition 1 (d-local algorithm): Given a network $G = (V, E)$, a problem specification ¶ for G, and an algorithm A that solves ¶. Algorithm A is said to be d-local if a node $n \in V$ executes a transition that requires the state of its d-hop neighbourhood to be queried.

4. Repair: A Corrector Protocol for Stabilizing Information Dissemination

In this section, we present a *corrector* protocol, called *Repair*. When *Repair* is composed with a fault-intolerant

[1] We will say two nodes m and n can collide if such a node p exists

dissemination protocol Σ, the resulting protocol (*Repair*$||\Sigma$) (pronounced Repair composed with Σ) is a stabilizing dissemination protocol, which guarantees that all nodes eventually download the correct code.

4.1 The *Repair* Protocol

Repair, shown in Figure 1, works with all reprogramming dissemination protocols that enable the detection of a fault. Specifically, Repair is triggered only when an erroneous state is detected due to a transient fault. Thus, any dissemination protocol that can interface with Repair must have enough state information to determine when a state is erroneous and, to the best of our knowledge, all current information dissemination protocols enable this. When a dissemination protocol detects a fault, then *Repair* is executed. In *Repair*, only corrupted nodes upgrade their code, avoiding unnecessary code updates.

Repair uses six special types of data packets (we call them *Repair* packets), in addition to dissemination packets used in the dissemination protocol:

- **Prob**: It contains code metadata and it is used to ask a neighbouring node to correct a fault.
- **Check**: It is used to request neighbouring nodes' code metadata.
- **Rep**: It contains a node's metadata and is used to reply to *Check* packet.
- **OK**: It is used to release some nodes from the correction process.
- **Cor**: It contains the correct metadata and is used to inform nodes about the correct metadata.
- **Hello**: It contains the correct metadata and is used to inform nodes that it has correct code.

Informally, *Repair* works as follows (The detailed algorithm is in Figure 3): When a node n_1 detects a fault after communicating with node n_2, it attempts to correct the fault. A *Prob* packet is sent to indicate a problem. If the fault cannot be corrected with n_2, then *Check* packets are broadcast, creating a *correction tree*, rooted at the node (n_1) that detected the fault. The leaf nodes of the tree responds to *Check* packets by sending *Rep* packets. If any of the leaf nodes detects a fault, it will spawn a subtree, within the main correction tree. Once a region in the network is reached where there is no fault, then no more subtree is spawned. This means that a node's neighbourhood have the same code version, with the version being presumed correct, since the probability of identical corruption of the metadata is very low. Then, this node responds through a *Rep* packet, and its subtree "disappears". Any node sending a *Rep* packet will cause its subtree to "disappear". Ultimately, *Cor* or *Hello* packet will be issued, with node n_1 getting the correct code. Notice that, since *Repair* ensures f-locality, the correction tree will be on depth f.

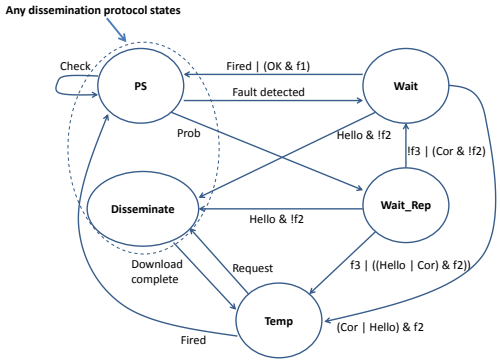


Fig. 1: The state machine. Two states in dashed area are the states of any dissemination protocol. f1=TRUE if sender of *OK* packet is the node which sent P, f2=TRUE if Sender.Vers=Receiver.Vers, f3=TRUE if all received metadata are the same.

4.2 Proof of Correctness of Repair

Lemma 1 (f-local stabilization): Given a network $G = (V, E)$, fault model F, and a set S of corrupted nodes, with the diameter of the corrupted area being f. Then, *Repair* guarantees that, eventually, all nodes in S will have the correct code.

Proof: Assume that a node n_f has detected a fault and wants to correct it. Given the diameter of the corrupted area is f, i.e., the maximum distance between two corrupted nodes in an area is f hops, we assume that all nodes on the path between n_{f-1} and n_0, namely n_{f-2}, n_{f-3},...,n_0 are corrupted. Node n_0 has neighbours with correct versions since corrupted area is of diameter f. We will prove by induction that node n_f will eventually get the correct code.
Base case: Since all neighbouring nodes of n_0 have correct metadata, according to *Repair*, n_0 will receive *Rep* packets with all equal metadata. Then n_0 requests the correct code from one of the neighbours and goes to *Disseminate* state to download the correct code.
Inductive hypothesis: Assume that a node n_i, where $0 < i < f - 1$, eventually receives *Hello* packet and updates its code from node n_{i-1}.
Inductive step: We need to prove that a node n_{i+1}, a neighbour node of n_i, eventually receives *Hello* packet and updates its code.

We know that in our protocol every node broadcasts *Hello* packet periodically upto h times after receiving *Hello* packet or updating its code. Now there are two cases that can happen after n_i receives *Hello* packet and updates its code.

- Case 1: n_{i+1} receives a *Hello* packet and updates its code from n_i, which proves the inductive step.
- Case 2: Because of message losses, n_{i+1} will not receive *Hello* packet from n_i. In this case, n_{i+1} waits *Wait_Time* and goes to PS state and starts to receive and send P packets. Eventually, n_{i+1} or a neighbour

PacketType $\in \{P, Prob, Check, Hello, OK, Rep\}$
// {6, 7, 8, 9, 10, 11} if used as timer IDs
Variables of process i**:**
state $\in \{1, 2, 3, 4, 5\}$ **Init** state:=1;
//for STABLE, WAIT, WAITREP, TEMP, DISSEMINATE

Collect:=12
version $\in \mathbb{N}$
firstProb $\in \{0, 1\}$ **Init** firstProb:=1
count $\in \mathbb{N}$ **Init** count:=0

PeriodProb: Timer
h: Timer
t: Timer
Send*Type*: Timer
*STATE*_Time: Timer

TableProb and TableRep: set of tuples {(id, version) : id $\in \mathbb{N}$, version $\in \mathbb{N}$} setTimer(X, Timer): timer with ID = X $\in$ [1, 12] set to Timer

Fig. 2: Variables of Repair algorithm.

state==1
case(upon⟨rcv (P, n, i)⟩)
state:=2
setTimer($Prob$, SendProb, n)
setTimer(*WAIT*, Wait_Time)
case(upon⟨rcv $(Check, n, i)$⟩)
setTimer(Rep, SendRep, n)
case(upon⟨rcv $(Prob, n, i)$⟩)
if(firstProb==1)
firstProb:=0
setTimer(*Collect*, SendProb+t)
else
TableProb ∪ $(n, n.$version)
endif
case(upon⟨rcv $(Prob, n,$ *ALL*)⟩)
if(i.version!=n.version)
state:=3
setTimer(*WAITREP*, WaitRep_Time)
setTimer($Check$, SendCheck, *ALL*)
endif
endcase

state==2
repeat every *PeriodProb*
send(*Prob*, i, n)
until (rcv.type!=*P*)
case(upon⟨rcv (OK, n, i)⟩)
state:=1
stopAllTimers()
TableProb:=∅
TableRep:=∅
case(upon⟨rcv (Cor, n, i)⟩)
if(i.version==n.version)
state:=4
setTimer(*TEMP*,Temp_Time)
else
bCast(*Cor*, i, *ALL*)
endif
case(upon⟨rcv $(Hello, n, i)$⟩)
if(i.version==n.version)
state:=4
setTimer(*TEMP*,Temp_Time)
else
state:=5
endif
endcase

state==3
case(upon⟨rcv (Cor, n, i)⟩)
if(i.version==n.version)
state:=4
setTimer(*TEMP*, Temp_Time)
else
bCast(*Cor*, i, *ALL*)
state:=2
setTimer(*WAIT*, Wait_Time)
endif
case(upon⟨rcv $(Hello, n, i)$⟩)
if(i.version==n.version)
state:=4
setTimer(*TEMP*, Temp_Time)
else
state:=5
endif
case(upon⟨rcv (Rep, n, i)⟩)
TableRep:= TableRep ∪ $\{(n, n.$version)$\}$
endcase

state==4
repeat every *SendHello*
bCast(*Hello*, i, *ALL*)
count:=count+1
until count>h
if(count>h)
state:=1
stopAllTimers()
TableProb:=∅
TableRep:=∅
endif
case(upon⟨rcv (Code Request, n, i)⟩)
state:=5
count:=0
endcase

if(timeout(*WAITREP*))
compare all n_j.versions ∈ TableRep
if(all and i.version are equal)
send(*Hello*, i, n)
state:=4
setTimer(*TEMP*, Temp_Time)
elsif(all are equal and i.version is not equal)
Request and download the code from n_j ∈ TableRep
send(*Hello*, i, n)
state:=4
setTimer(*TEMP*, Temp_Time)
else
bCast(*Prob*, i, *ALL*)
state:=2
setTimer(*WAIT*, Wait_Time)
endif
endif

if(timeout(*Collect*))
compare all n_j.versions ∈ TableProb
if(all are equal)
bCast(*OK*, i, *ALL*)
state:=4
setTimer(*TEMP*, Temp_Time)
else
state:=3
setTimer(*WAITREP*, WaitRep_Time)
setTimer($Check$, SendCheck, *ALL*)
endif
endif

case: state==5
send/download the code.
state:=4
setTimer(*TEMP*,Temp_Time)
endcase

if(timeout(PacketType, j))
if(j==*ALL*)
bCast(PacketType, i, *ALL*)
else
send(PacketType, i, j)
endif
endif

if(timeout(*TEMP*))
state:=1
stopAllTimers()
TableProb:=∅; TableRep:=∅
endif

if(timeout(*WAIT*))
state:=1
stopAllTimers()
TableProb:=∅; TableRep:=∅
endif

Fig. 3: Repair Algorithm.

node of n_{i+1} will detect the fault, and executes *Repair* again, this time the affected area will be of size $(f - i)$, assuming no further fault has occurred. Assuming that the number of message losses is finite, eventually, n_{i+1} will get a *Hello* packet. The first case will be eventually true.

□

Theorem 1 (Correctness of Repair*):* Given a network $G = (V, E)$, transient fault model F, a F-intolerant information dissemination protocol Σ for a specification σ. Then, *Repair* is a *corrector component* of Σ for σ.

Proof: The proof follows from Lemma 1, and from the fact that Repair is only triggered when a fault is detected. □

Observe also that *Repair* is *self-correcting*, i.e., if a transient fault occurs in *Repair* when *Repair* is executing, leading to a node downloading the wrong code. This fault will eventually be detected, and *Repair* will be executed again to correct the fault. Put otherwise, *Repair* is a corrector component for *Repair* itself.

5. Experimental Setup

We perform TOSSIM[11] simulations on a 20x20 grid network to evaluate *Repair*. We set the distance between neighbouring nodes to 10 feet. Each node has a communication radius of around 30 feet. Network topology with asymmetric links is constructed by a tool given on tinyos.net. Each node is given a noise model from a heavy-meyer noise trace file.

Parameter values used in our simulation are given in Table 1. Some of the parameter values depend on other

parameters. For example, *WaitRep_Time* is the time for waiting for *Rep* packets after broadcasting *Check* packet. So, *WaitRep_Time* $\geq$ *SendCheck+SendRep*. *Wait_Time* should be set according to the code size and the size of the network. If the network and code size is large, this time should be large enough to allow neighbouring nodes to correct their code and forward it. Usually nodes enter *Temp* state from *Wait* state where it waits less time. The only case when a node waits *Wait_Time* is when there is a packet loss. *Temp_Time* time does not depend on other parameters. The value of t should be small because a node waits a maximum *SendProb* time to receive all possible *Prob* packets. The values of h and p can be set to any values.

In our simulations, each node periodically broadcasts P packet, with period randomly selected between $[0, U]$ at the start. We simulated two scenarios: (i) we incremented the number of corrupted nodes per circular area, which has diameter of 60 feet, and (ii) we kept the number of corrupted nodes to 5 and increased the size of a given (square) area, i.e., decrease the fault density. In both scenarios, the corrupted nodes were selected randomly in the given area. We then counted (i) the number of packets, (ii) the number of involved nodes, i.e., nodes that sent a *Repair* packet, and (iii) the number of nodes which changed their states to *Wait* and/or *WaitRep* states. For each given number of corrupted nodes in the first scenario and for each length of square area in the second scenario, we ran simulations 5 times and computed the min, average and max values.

6. Simulation Results

In this section, we present two metrics to show the *locality property* of *Repair*, namely (i) number of nodes executing *Repair*, and (ii) number of packets sent.

Number of nodes: From Figure 4(a), we observe that, on average, the number of nodes executing the protocol varies linearly with the number of corrupted nodes. Given that the number of nodes involved is much less than the size of the network, it indicates that the number of nodes involved is proportional to the size of the corrupted area. Further, in Figure 4(b), we observe that, when 5 transient faults were injected, the number of nodes executing the protocol becomes almost constant, on the average. This is because, with decreasing fault density, the transient faults appear as single independent faults, with each of them involving a similar number of nodes, i.e., since the area increases, the chance of the 5 faulty nodes being neighbours is very low. These two observations support the fact that *Repair* in f-local, where f is the diameter of the fault-affected area.

Number of packets: We observe a similar trend as in Figure 5, further supporting the f-locality property of *Repair*.

7. Composition of *Repair* with Varuna

In this section, we discuss the composition of Repair with an existing information dissemination protocol, namely Varuna [3]. The reason for choosing Varuna is that it is one of the latest information dissemination protocol that have been proposed.

As mentioned before, Repair is triggered by the detection of a fault. In Varuna, such a detection is enabled by one of the following conditions: (i) two nodes' metadata (i.e., version number) are corrupted in such a way that the difference in versions is greater than 1, and (ii) the receiver of an advertisement message finds that its version is bigger than the advertised one and, at the same time, the sender exists in its neighbourhood table.

We simulated the composite protocol of Varuna and Repair in TOSSIM. All nodes, except faulty nodes, are booted in the first minute. Faulty nodes are located at the center of the network. A packet with new version number is injected at 2 minutes. We simulated three faulty scenarios: (i) with 1 fault, (ii) with 4 faults and (iii) with 7 faults. For each scenario, we booted the faulty nodes (i) 30 seconds, (ii) 45 seconds and (iii) 60 seconds after injecting the new code, so that only a proportion of nodes had the new code version. We are specifically interested in (i) the overhead induced by *Repair* on the performance of Varuna and (ii) the number of nodes with correct code at a given time. We simulated Varuna in conditions similar to those detailed in Section 5. Further, the values for Varuna-specific parameters are: DISS-RAND=2 sec, ADV-RAND=2 sec, τ=8 sec, T_{MOODY}=1 min. For reasons of space, we only present a sample of the results obtained.

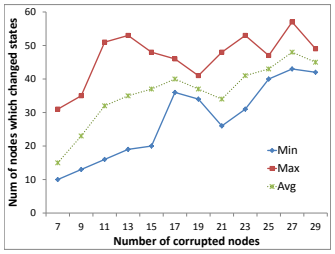
Performance of Repair: As can be observed in Figures 7 and 8, in all cases, injecting transient faults in the network during Varuna only execution causes the whole network to disseminate stale code. On the other hand, when Varuna is composed with Repair, every node downloads the correct code.

Packet Overhead: In Figure 6(a), it can be seen that the packets overhead induced by Repair on Varuna is low. Specifically, with 7 faulty nodes, the packet overhead is less than 3%. From Figure 5, it can be observed that the number of packets will increase linearly with increasing number of corrupted nodes. The reason for the linear increase (as opposed to a constant value) is that the fault density increases when more corrupted appear at the centre of the network (condition under which we simulated the composite protocol).

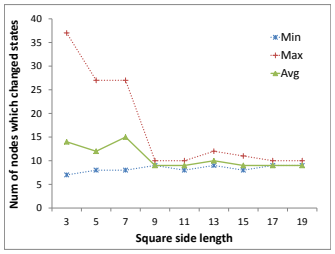
Temporal Overhead: In Figure 6(b), it can be observed that the whole network receives the new code in approximately 80 seconds, after the new code has been injected into the network. Further, it can be observed that, when there are faulty nodes in the network, the time for the whole network to receive the correct code is approximately 80 seconds. Thus, *there is almost no temporal overhead induced by Repair on Varuna*, highlighting the composable nature of Repair. This is so because Repair executes in parallel with Varuna, and also corrects only corrupted nodes.

Table 1: Parameters for the simulation

Wait_Time	50 sec	*SendRep*	2 sec	*SendHello*	1.5 sec	*Temp_Time*	30 sec
SendCheck	2 sec	*WaitRep_Time*	4 sec	*SendProb*	1 sec	t	0.2 sec
U	60 sec	h	2	*PeriodProb*	7 sec	p	5

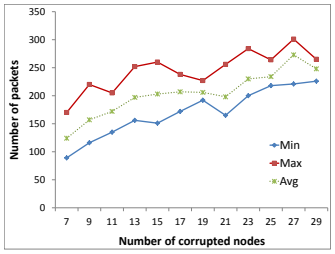


(a) Num. of nodes executing *Repair* vs Num. of Corrupted Nodes, Area diameter = 60ft

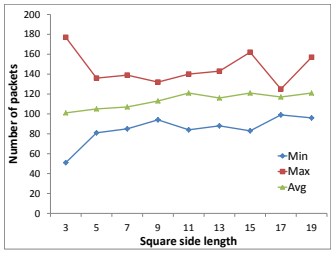


(b) Num. of nodes executing *Repair* for 5 corrupted nodes per sq. area.

Fig. 4

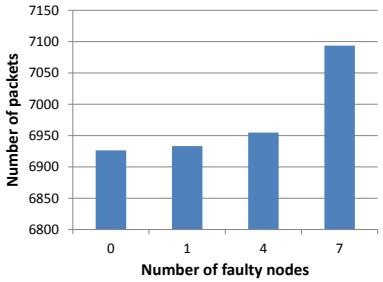


(a) Num. of packets vs Num. of corrupted nodes, Area diameter = 60ft

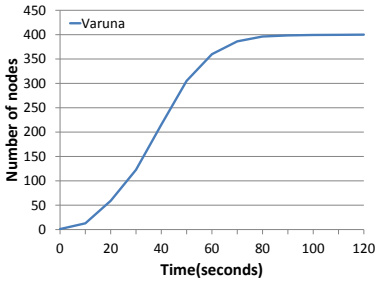


(b) Num. of packets for 5 corrupted nodes per sq. area.

Fig. 5



(a) Number of (Repair + ADV) packets sent vs number of corrupted nodes



(b) Completion time of Varuna (no transient faults)

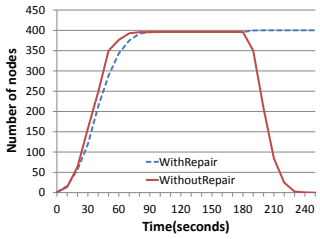
Fig. 6: Varuna || Repair : Network Size: 20 * 20, Nodes Corrupted at Random

8. Conclusion and Further Work

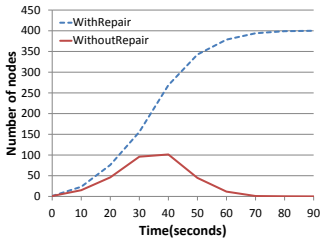
We have presented *Repair*, which when integrated with a fault-intolerant information dissemination protocol, transforms it into a stabilizing fault-tolerant one. We have shown the performance of Repair, and also when it was integrated with Varuna, one of the most recent information dissemination protocol. In presence of faults, Varuna causes all nodes to download the wrong code, while the composite protocol of Varuna and Repair ensured that all nodes eventually download the new code, while incurring minimal overhead.

References

[1] P. E. Lanigan, R. Gandhi, and P. Narasimhan, "Disseminating code updates in sensor networks: Survey of protocols and security issues," Institute for Software Research, Carnegie Mellon University, Tech. Rep. CMU-ISRI-05-122, Oct. 2005.

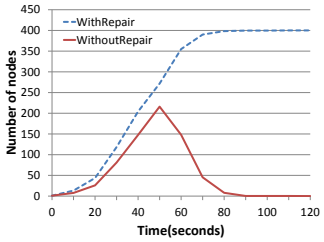


(a) Number of nodes which receive correct code vs Time

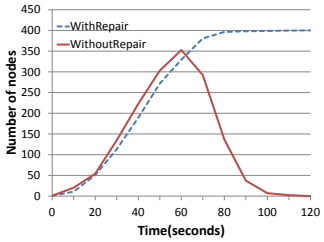


(b) Number of nodes which receive correct code vs Time

Fig. 7: Varuna and Varuna || Repair: (a) 4 faulty nodes are booted 180 seconds after new code injection(at 5 minutes). (b) 7 faulty nodes are booted 30 seconds after new code injection(at 2 minutes 30 seconds).



(a) Number of nodes which receive correct code vs Time



(b) Number of nodes which receive correct code vs Time

Fig. 8: Varuna and Varuna || Repair: (a) 7 faulty nodes are booted 45 seconds after new code injection(at 2 minutes 45 seconds) (b) 7 faulty nodes are booted 60 seconds after new code injection(at 3 minutes).

[2] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*. Berkeley, CA, USA: USENIX Association, 2004, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251175.1251177

[3] R. K. Panta, M. Vintila, and S. Bagchi, "Fixed cost maintenance for information dissemination in wireless sensor networks," in *Proc. SRDS*, 2010, pp. 54–63.

[4] S. S. Kulkarni and L. Wang, "Mnp: Multihop network reprogramming service for sensor networks," in *In Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS*, 2005, pp. 7–16.

[5] L. Mottola, G. Picco, and A. A. Sheik, "Figaro: Fine-grained software reconfiguration for wireless sensor networks," in *Proceedings of EWSN*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 286–304.

[6] K. N. et al., "Sensor networks data fault types," *Transactions on Sensor Networks*, vol. 5, no. 3, 2009.

[7] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proc. of* 7^{th} *Symp. on Operating Systems Design and Implementation (OSDI)*, 2006.

[8] N. Finne, J. Eriksson, A. Dunkels, and T. Voigt, "Experiences from two sensor network deployments self-monitoring and self-configuration keys to success," in *Proc. of Int. Conf. on Wired/Wireless Internet Communications (WWIC)*, 2008.

[9] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, ser. SenSys '04. New York, NY, USA: ACM, 2004, pp. 81–94. [Online]. Available: http://doi.acm.org/10.1145/1031495.1031506

[10] S. Dolev, *Self-Stabilization*. MIT Press, 2000.

[11] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: accurate and scalable simulation of entire tinyos applications," in *Proceedings of the 1st international conference on Embedded networked sensor systems*, ser. SenSys '03. New York, NY, USA: ACM, 2003, pp. 126–137. [Online]. Available: http://doi.acm.org/10.1145/958491.958506

[12] I. Crossbow Technology, "Mote in-network programming user reference, www.tinyos.net/tinyos-1.x/doc/xnp.pdf," 2003.

[13] T. Stathopoulos, J. Heidemann, and D. Estrin, "A remote code update mechanism for wireless sensor networks," UCLA, Center for Embedded Networked Computing, Tech. Rep. CENS-TR-30, 2003.

[14] N. Reijers and K. Langendoen, "Efficient code distribution in wireless sensor networks," in *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, ser. WSNA '03, New York, NY, USA, 2003, pp. 60–67. [Online]. Available: http://doi.acm.org/10.1145/941350.941359

[15] J. Jeong and D. Culler, "Incremental network programming for wireless sensors," in *IEEE Sensor and Ad Hoc Communications and Networks (SECON*, 2004, pp. 25–33.

[16] O. Gnawali, K. Jang, J. Paek, M. Vieira, R. Govindan, B.Greenstein, A.Joki, D.Estrin, and E. Kohler, "The tenet architecture for tiered sensor networks," in *SenSys*, 2006, pp. 153–166.

[17] G. Tolle and D. E. Culler, "Design of an application-cooperative management system for wireless sensor networks," in *EWSN*, 2005, pp. 121–132.

[18] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler, "Marionette: using rpc for interactive development and debugging of wireless embedded networks," in *Proc. IPSN*, 2006.

[19] A. Arora and S. Kulkarni, "Detectors and correctors: A theory of fault-tolerance components," in *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, May 1998.

Whitewash-Aware Reputation Management in Peer-to-Peer File Sharing Systems

Xiao YU[1] **and Satoshi FUJITA**[1]
[1]Department of Information Engineering, Hiroshima University
Higashi-Hiroshima, 739-8527, Japan

Abstract— *In this paper, we propose new schemes for the reputation management in P2P applications which discourage whitewash while encouraging good behaviors. The basic idea of the schemes is to design update rules for the reputation scores to satisfy the following requirements: 1) the score of a peer is strictly greater than the initial score at any point in time if it conducted at least one good behavior, 2) the score gradually increases if it conducted a good behavior while it rapidly decreases if it conducted a bad behavior, and 3) the strength of penalty is refined by allowing the system to give a penalty for several consecutive rounds.*

1. Introduction

Peer-to-Peer systems (P2Ps, for short) are autonomous distribute systems which have been used in many applications such as file sharing, video streaming, IP phone, and others. Different from traditional Client/Server (C/S) systems which rely on few dedicated servers, services in P2Ps are provided by each computer participating in the system in a "peer-to-peer" manner. In other words, each computer in P2Ps, called **peer** hereafter, plays the role of a client and a service provider at the same time. Such a remarkable property of P2Ps enables the designer of distributed systems to increase the scalability and the fault-tolerance of the constructed system, because it effectively removes the single point of failure existing in C/S systems as well as the service bottleneck.

However, such a distributed nature of P2Ps would cause several critical issues, such that a malicious peer can provide wrong, devastating services to the client peers, the quality of the services is not guaranteed by any authority, and the security of transactions could not be retained. In this paper, we focus on P2P reputation systems as a way of resolving such issues. In typical reputation management systems, each recipient of a service can *evaluate* the quality of the service and a collection of such evaluations will be disclosed to all participants so that it could be used to select safe and appropriate services in the next time. Examples of reputation systems include [2], [3], [4], [12], [13], [14]. A key idea of such reputation systems is to share information on past transactions among all participants to the system, i.e., if a transaction conducted by peer i is observed by peer j and another transaction conducted by i is observed by peer k ($\neq j$), by merging those two observations, we will have a more reliable evaluation concerned with the transactions conducted by peer i than the case in which each peer individually keeps such an evaluation.

In many P2P applications, a peer to have a high reputation will be granted to access high quality services such as the broader communication bandwidth and the video streaming in HD quality. On the other hand, the reputation of a peer rapidly becomes worse if it conducted malicious actions, such as an intentional provision of low quality services and the distribution of malwares such as spyware and computer viruses. In other words, the reputation system works as an incentive mechanism for the participants to conduct good behaviors. However, such an effect of reputation systems can be significantly reduced if a peer with a bad reputation could become a new participant by changing its identifier (ID) after leaving the system. Such a malicious behavior of peers is known as **whitewash**, and it has been recognized as a crucial issue in many distributed applications with reputation management [6], [5], [9].

In this paper, we propose new schemes for the reputation management in P2P applications *which discourage whitewash while encouraging good behaviors*. The basic idea of the schemes is to design update rules for the reputation scores to satisfy the following three requirements: 1) the score of a peer is strictly greater than the initial score at any point in time if it conducted at least one good behavior, 2) the score gradually increases if it conducted a good behavior while it rapidly decreases if it conducted a bad behavior, and 3) the strength of penalty is refined by allowing the system to give a penalty during several consecutive rounds.

The remainder of this paper is organized as follows. After overviewing related works in Section 2, Section 3 describes the model of P2P reputation systems. Section 4 proposes basic update rules for reputation scores, which is extended in Section 5. Section 6 proposes several reputation management schemes based on the extended update rules. Finally, Section 7 concludes the paper with future work.

2. Related Work

There are few proposals on whitewash-aware reputation management in spite of the importance of the problem. Pinninck *et al.* proposed a scheme which increases the

resistance of trust assessment schemes against whitewash attacks with the aid of social networks [7]. This scheme assumes that all interactions among peers are conducted according to the following simple protocol: 1) the initiator peer p chooses a set of potential partner peers S_p and evaluates the trust of all members in S_p; 2) p selects a peer q in S_p and sends an invitation message to q, 3) if q accepts the invitation, it starts an interaction with p, 4) after completing the interaction, q sends a feedback about the interaction to p. The key idea of the scheme is to use a social network in which each peer must be adjacent with a set of contact peers. Invitation messages are routed to the receivers through such contact peers, so that any peer wishing to interact with other peers must know at least one contact peer in the social network. Such a restriction makes a simple whitewashing meaningless, since if it changes ID, contact peers do not recognize the peer any more, so that the invitation message will not be routed to any receiver (note that the scheme could not completely prohibit whitewash if each peer can have several temporary IDs and tries to connect the network through a permanent ID among them).

Chen *et al.* proposed a scheme to identify whitewashers in P2P file sharing systems using the notion of observation preordering [1]. This scheme is based on an assumption such that actions conducted in our daily life are *habitual* so that it is hard to change even under different situations. Whitewashers are no exceptions. Namely, even after re-entering the system with a different ID, a whitewasher should contact similar peers to download files in a similar category. Observation preordering is a data structure to record the history of actions concerned with a peer, which is observed and recorded by another peer during the interaction with the target peer. Thus, for example, after interacting with peer j, peer i stores (or updates) the observation preordering concerned with j in its local storage. Suppose that j is malicious and conducts a whitewash to acquire new ID k ($\neq j$). By the assumption described above, peer k should contact peer i again to download files, and such an action is observed by i which will be stored as an observation preordering concerned with k. Thus, peer i could identify that k is likely to be j by comparing observation preorderings concerned with j and k, and if it concludes that k is j, it recognizes k as a malicious peer and degrades the reputation score of k accordingly.

How to encourage peers to conduct collaborative actions is another important issue in realizing practical incentive systems. Tseng and Chen proposed a free-rider aware reputation scheme for P2P file-sharing systems [11]. In this scheme, peers and files are divided into five levels depending on the reputation score, and the incentive mechanism is designed in such a way that a peer which does not share its files with the other peers can not access files at a higher level; i.e., in order to access files at a higher level, it needs to share its files with the other participants. This scheme also provides a penalty mechanism such that: 1) if a peer tried to share harmful files with the other peers including malwares and inauthentic files, and if such a malicious behavior is reported by the other peers, the reputation of the peer is reduced according to the penalty function (hence the level of the peer would also degrade accordingly), and 2) if a peer shares no files with the other peers, the reputation score gradually decreases as the elapsed time increases.

3. Model

In this section, we describe the model of P2P systems considered in this paper. The model of malicious actions of peers and the basic framework of reputation management will also be described.

3.1 System Model

In this paper, we consider P2P file sharing systems consisting of a number of peers which play the role of a client and a service provider at the same time. Each peer holds several files which can be shared with the other peers. Each peer, which wishes to acquire a copy of a file, firstly sends an inquiry message to the system so that the inquiry message will be delivered to peers holding the requested file [8], [10], [15]. The requesting peer will receive a response from several peers holding the requested file, and the receiver conducts the selection of a peer from the set of candidate peers according to the *reputation* of the candidates; e.g., high reputation peers are likely to be selected as an uploader compared with low reputation peers. Download of the requested file is conducted merely from the selected peer.

After completing the download, the downloader evaluates the transaction and gives a score to the uploader so that it reflects the *degree of satisfaction* of the downloader concerned with the transaction. In other words, the score is given for each transaction even if such transactions are provided by the same uploader. Such scores are aggregated to a central manager which keeps the reputation scores of all peers in the system, and if a peer conducts an evaluation of another peer, the outcome of the evaluation is immediately notified to the central manager.

3.2 Reputation Score

The **reputation score** of a peer is a sum of scores given by the downloaders. In this paper, we assume the existence of an appropriate incentive mechanism so that a peer with high reputation score will be granted a right to access high quality services, such as the higher priority while conducting a download from service providers and a wider bandwidth when it uses shared communication channels. Thus, it is natural to assume that *every rational peer should try to increase its reputation score*. If it is an honest peer, such an increase of the score will be attained by providing satisfactory transactions to the downloaders, but if it is dishonest, it tries to cheat by conducting malicious actions,

such as the issue of incorrect report to decrease the score of other peers, refusal of given requests, and provision of low quality services instead of providing requested services.

In general, to encourage honest actions of the peers, reputation scores should be managed in such a way that: 1) the score of a peer increases if it conducted collaborative actions to increase the satisfaction of downloaders (e.g., to increase the score by Δ^+), and 2) the score decreases if it conducted adversarial actions to decrease the satisfaction of downloaders (e.g., to decrease the score by Δ^-). As the strength of the penalty increases, i.e., as the value of Δ^- increases, each peer would likely to conduct collaborative actions without conducting adversarial actions, i.e., an incentive to encourage collaborative actions works well. However, if it was too strong, a peer which conducted an adversarial action would select a (malicious) way such that it quits the system once and re-enters the system as a new participant. Such a malicious behavior of a peer is called **whitewash** which is known to degrade the effectiveness of the underlying incentive mechanisms. In fact, to discourage whitewash, Δ^- must not be too large, but if it is not too large, the force to encourage honest actions should become weak.

4. Basic Update Rule

In this section, we propose a collection of update rules of the reputation scores which discourage whitewashes but encourage honest actions. The proposed rules are designed to satisfy the following requiremenrs:

1) The reputation score of a peer increases if it conducts a collaborative action, while it decreases if it conducts an adversarial action.
2) The reputation score is strictly larger than the initial score at any point in time, if it conducted at least one collaborative action.

The second requirement intends that a peer conducted collaborative actions becomes harder to be penalized even if it occasionally conducts adversarial actions.

4.1 Update Rules

Let $R_i \in (0,1)$ denote the reputation score of peer i. R_i is initialized to R_0 at the time of participation. Suppose that peer j downloaded a file from peer i, and j is satisfied with the transaction. Then, peer j notifies the result of such a positive evaluation to the central manager, and after receiving it, the central manager updates the reputation score of i as follows:

$$R_i \;:=\; \alpha R_i + (1-\alpha) \tag{1}$$

where α is a parameter in range $(0,1)$. The above update rule indicates that as the value of R_i increases, the "amount of increase" gradually decreases even if it repeatedly conducts collaborative actions, e.g., if $R_0 = 0$, a sequence of collaborative actions monotonically increases the score as

$$0 \to (1-\alpha) \to (1-\alpha^2) \to (1-\alpha^3) \to \cdots .$$

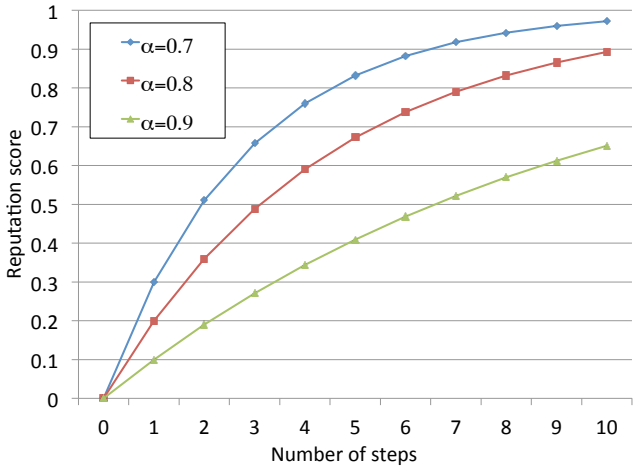


Figure 1: Increase of the reputation score along with collaborative actions ($R_0 = 0$).

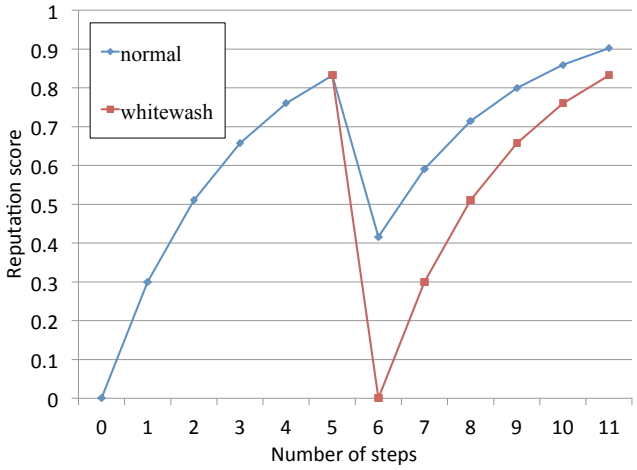


Figure 2: Badness of the reputation score under whitewash.

Figure 1 illustrates the increase of the reputation score along with collaborative actions, for different α's. On the other hand, if j is not satisfied with the transaction, j sends a negative notification to the central manager, and after receiving it, the central manager updates the reputation score of i as follows:

$$R_i \;:=\; \frac{R_i - R_0}{\beta} + R_0 \tag{2}$$

where β is a parameter greater than 1. The reader can easily verify that the second condition described above is certainly satisfied for any selection of $\beta > 1$. In fact, once $R_i > R_0$ holds, this inequality remains to hold even after any number of applications of the second update rule.

4.2 Analysis

In the last section, we observed that by conducting a whitewash, the reputation score becomes worse than the score *immediately before* the whitewash. In this section, we extend this simple argument. More concretely, we prove that by conducting a whitewash, the reputation score always becomes worse than the case without whitewash for any sequence of collaborative and adversarial actions. Let S be a ternary string representing a sequence of actions, where 0 and 1 indicate collaborative and adversarial actions respectively, and 2 indicates whitewash. Let $R(S)$ denote the reputation score of a peer after conducting an action sequence S.

We can prove the following claim.

Remark 1: Let $S = a_1, a_2, \ldots, a_n$ be a sequence of actions conducted by a user starting with a collaborative action, and S' be a sequence of actions which is obtained from S by inserting a whitewash after the i^{th} action for some $1 \leq i \leq n$. Then, $R(S) > R(S')$.

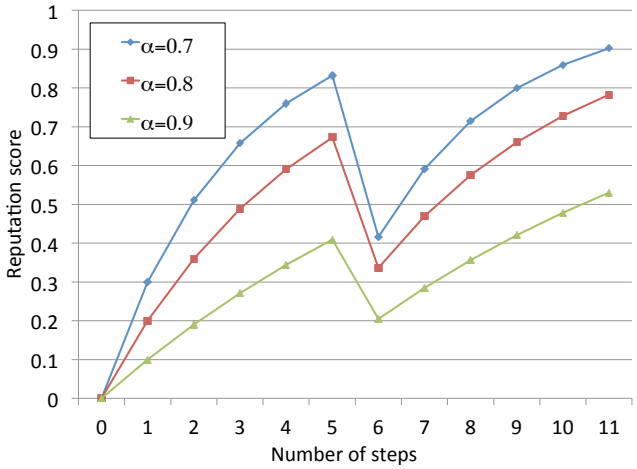
Proof: Suppose that a whitewash is inserted after the i^{th} action, i.e., it divides S into two parts $S_1 = a_1, a_2, \ldots, a_i$ and $S_2 = a_{i+1}, \ldots, a_n$. Since a whitewash initializes the reputation score, $R(S') = R(S_2)$. By the second condition described above, since it is assumed that S_1 contains at least one collaborative action, $R(S_1) > R_0$. By the definition of update rules, as the initial score R_0 increases, the resultant score monotonically glows. Hence the claim follows. ■

The badness of whitewash with respect to the reputation score is illustrated in Figure 2.

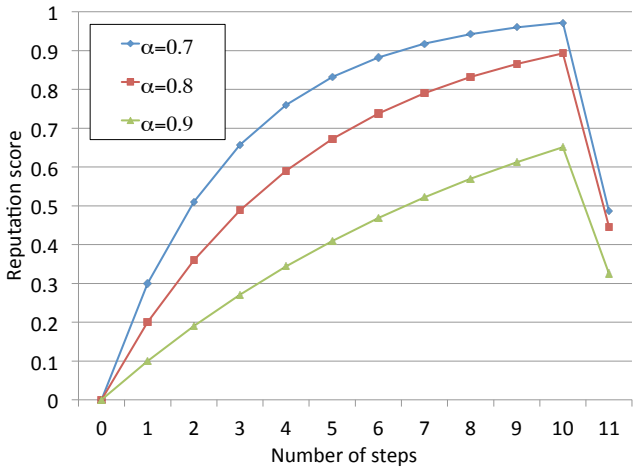
5. Extension

5.1 Motivation

In the above scheme, each peer who conducted an adversarial action is penalized by "uniformly" reducing the reputation score to $1/\beta$. Although it certainly penalizes adversarial actions of malicious peers, those rules give a penalty exactly once. In other words, after reducing the reputation score, the system "allows" the peer and treats him as an honest peer in the succeeding rounds. Thus, it could not effectively work as a *deterrent for addicts* of adversarial actions of malicious peers particularly when they repeat a sequence of actions consisting of an adversarial action and few collaborative actions. For example, if $\alpha = 0.5$, $1/\beta = 0.6$, and $R_0 = 0$, by repeating three collaborative actions after the participation, the score of the peer becomes $1 - 0.5^3 = 0.875$, and by conducting an adversarial action at that time, the score reduces to $0.6 \times (1 - 0.5^3) = 0.525$, but it is slightly larger than the score after the first collaborative action. In other words, one penalty is weaker than two collaborative actions in this case. On the other hand, the penalty seems to be too strong for the peers which have repeated many collaborative actions. For example, if it repeats 1000 collaborative actions in the above example,



(a) Adversarial action at the sixth step.



(b) Adversarial action at the 11th step.

Figure 3: Difference of the impact of adversarial actions to the reputation score.

the penalty for (only) one adversarial action is heavier than 998 collaborative actions. Figure 3 shows the change of the reputation score according to the difference of the position of an adversarial action in a sequence of collaborative actions, assuming $R_0 = 0$ and $1/\beta = 0.5$. When an adversarial action occurs at the sixth step, it "cancels" three or four collaborative actions conducted before it (Figure 3 (a)). However, if it occurs at the 11th step, it cancels 6 steps for $\alpha = 0.9$ and 8 steps for $\alpha = 0.7$ (Figure 3 (b)), which is larger than the case of the sixth step.

Such an unbalance on the number of consecutive collaborative actions which are comparable to one adversarial action should be overcome by reducing β as small as possible (i.e., the difference becomes small by decreasing β), and by introducing an additional mechanism for the penalization. The time transition of the reputation score for different β's is illustrated in Figure 4. It could be observed that the reduction

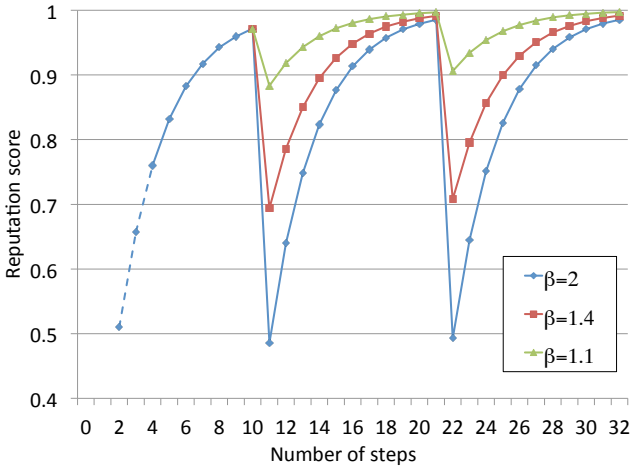


Figure 4: Time transition of the reputation score for different β ($\alpha = 0.7$ and $R_0 = 0$).

of the score significantly decreases as β approaches to one.

5.2 Scheme

Our main idea for the improvement of the basic scheme is to reduce the amount of increase of the reputation score during n consecutive rounds after detecting an adversarial action, where n is a parameter determined later. More concretely, we use the following rule instead of Equation (1) during n consecutive rounds after encountering an adversarial action:

$$R_i \ := \ \gamma R_i + (1 - \gamma) \tag{3}$$

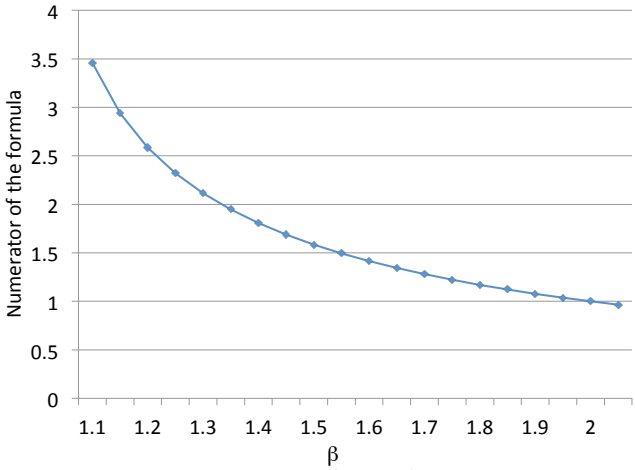
for some $\gamma > \alpha$. The reader should note that if n is too small, it does not effectively frighten peers to conduct adversarial actions, whereas if n is too large, it will encourage adversarial peers to conduct a whitewash. Thus an appropriate value of parameter n should be calculated carefully, which should depend on parameters α, β, γ, and the value of R_i at the time of encountering an adversarial action.

5.3 Analysis

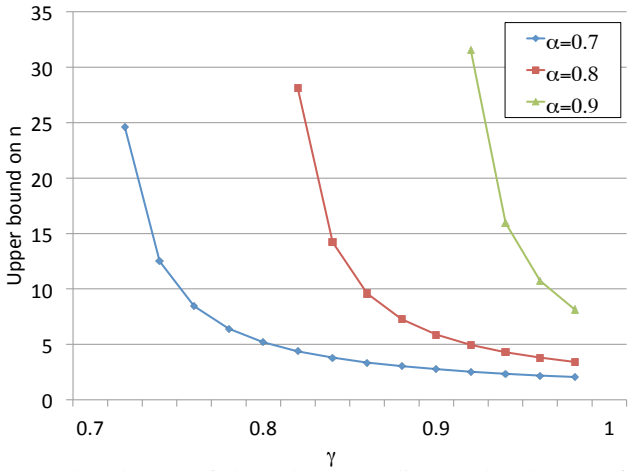
In this section, we derive an upper bound on parameter n in the sense that if it exceeds the value, it works as an incentive to conduct a whitewash. Recall that $R(S)$ denotes the reputation score after conducting an action sequence S which is represented by a ternary string in such a way that 0 and 1 indicate collaborative and adversarial actions respectively, and 2 indicates whitewash.

The following claim is easy to prove since the effect of whitewash will be maximized if it is conducted immediately after an adversarial action.

Remark 2: Let $S = a_0, a_1, a_2, \ldots, a_n$ be a sequence of actions such that $a_0 = 1$ and $a_i = 0$ for $1 \leq i \leq n$, S' be a sequence of length $n + 2$ which is obtained from S by "inserting" a whitewash at the second position. Note that in



(a) Numerator of Equation (4).



(b) The change of the value according to the change of parameters α and γ ($\beta = 2$).

Figure 5: Upper bound on n.

sequence S, n consecutive actions are penalized by reducing the increase of the reputation score. Then, the extended scheme does not encourage whitewash if $R(S) > R(S')$.

Let x be the score before conducting action sequence S. Then, we have

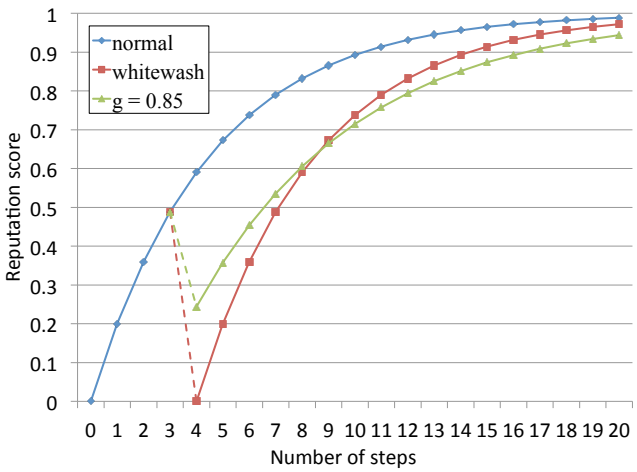
$$R(S) \ = \ \gamma^n \times \left(\frac{x - R_0}{\beta} + R_0 \right) + 1 - \gamma^n$$

and

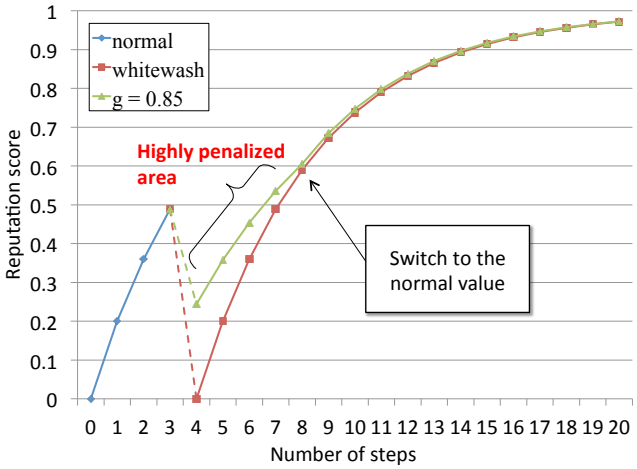
$$R(S') \ = \ \alpha^n \times R_0 + 1 - \alpha^n$$

Thus, in order to satisfy $R(S) > R(S')$, we should have

$$\gamma^n \times \left(\frac{x - R_0}{\beta} + R_0 - 1 \right) \ > \ \alpha^n \times (R_0 - 1)$$



(a) When it does not switch to Rule (1).



(b) When it switches to Rule (1) after four times of repetitions.

Figure 6: The role of parameter n.

that is,

$$\left(\frac{\alpha}{\gamma}\right)^n > \frac{1 - R_0 - \frac{x - R_0}{\beta}}{1 - R_0} = 1 - \frac{x - R_0}{\beta(1 - R_0)} > 1 - \frac{1}{\beta}$$

where the last inequality is due to $x < 1$. By taking a logarithm, we have

$$n \log(\alpha/\gamma) > \log(1 - 1/\beta)$$

Since $\alpha < \gamma$, $\log(\alpha/\gamma) < 0$. Thus,

$$n < \frac{\log \beta - \log(\beta - 1)}{\log \gamma - \log \alpha}. \quad (4)$$

The numerator of Equation (4) gradually decreases as β increases, as is shown in Figure 5 (a). In addition, for a fixed α, the right hand side of the formula decreases as γ increases from α, as is shown in Figure 5 (b) (in this figure, we fix β to two). By this figure, we can see that we could apply Rule (3) at most six times if parameters are determined as $\alpha = 0.7$, $\beta = 2$, and $\gamma = 0.78$, but it decreases to four times if we slightly increase γ to 0.82.

An example of the time transition of the reputation score is illustrated in Figure 6. This figure assumes $\alpha = 0.7$, $\beta = 2$, and $\gamma = 0.85$. If we apply Rule (3) instead of Rule (1) forever, as is shown in Figure 6 (a), the score after whitewash eventually becomes larger than the penalized score. However, by switching the rule to Rule (1) after passing an appropriate number of repetitions (e.g., in this example, by switching the rule after three times of applications), we can guarantee that the resulting score is still greater than the score after whitewash, as in shown in Figure 6 (b).

6. Schemes

In this section, we propose several reputation management schemes based on the extended update rules.

6.1 Threshold Type

The first idea is to switch the rule from Rule (3) to Rule (1) by the value of the reputation score. More concretely, it switches the rule when: 1) it encounters the upper bound on n, or 2) the reputation score exceeds a predetermined threshold (e.g., 0.8). This scheme is intended to "allow" users when their reputation score exceeds the threshold, since the fact of exceeding the threshold indicates that it has repeated sufficient number of collaborative actions. In fact, since the score after applying Rule (2) is at most $1/\beta$, to reach threshold θ ($> 1/\beta$), it should repeat at least m collaborative actions satisfying the following inequality:

$$\theta < \gamma^m(1/\beta) + 1 - \gamma^m.$$

By solving it, we have the following lower bound on m,

$$m > \frac{\log(1 - \theta) - \log(1 - 1/\beta)}{\log \gamma}.$$

6.2 Counting Type

The second idea is to (gradually) increase the number of repetitions depending on the number of adversarial actions which have been conducted by the corresponding peer. More concretely, the scheme works as follows: 1) Prepare a variable w to count the number of adversarial actions conducted by the peer. Variable w is initialized to zero and is incremented when it conducted an adversarial action. 2) The number of penalizations (i.e., the number of applications of Rule (3)) is determined as

$$\min\{f(w), n^*\}$$

where f is an appropriate monotonically increasing function such as $f(w) = w$ and $f(w) = w^2$, and n^* is an upper bound on n determined by Equation (4).

6.3 Random Type

The third scheme uses the notion of randomization. In the last two schemes, each peer can predict the strength of penalization from the outcome of past trials. For example, in the first scheme, a malicious peer knows that the penalization finishes after reaching its score to the threshold, and in the second scheme, a malicious peer knows from its experience that the strength of penalization against its next adversarial action. To effectively hide such information from malicious peers, a randomization could be used in the following manner: 1) After detecting an adversarial action of a peer, the central manager selects a random number r from set $\{1, 2, \ldots, n^*\}$. 2) It then penalizes during r consecutive rounds after reducing the score of the corresponding peer by Rule (2).

7. Concluding Remarks

In this paper, we propose new schemes for the reputation management in P2P applications which discourages whitewash while encouraging good behaviors. Our proposed scheme can control the strength of penalty against adversarial actions.

Topics for our future work are listed as follows:

- The evaluation of the proposed schemes considering the incentive of users to participate in the system. In actual P2Ps, each user reserves a right to leave from the system if she feels that it is not attractive compared with the required cost. Our current analysis misses such an issue.
- Combination with other techniques to discourage whitewash. For example, by combining the proposed schemes with Adrian and Marco's scheme described in Section 2, we could reduce the number of whitewashes in actual P2P environments.
- Detailed analysis of the fairness in the proposed schemes. We need to give a formal definition of fairness, as well as the tuning of several parameters to meet the fairness criteria.

Acknowledgements

This work was supported in part by the Scientific Grant-in-Aid from Ministry of Education, Science, Sports and Culture of Japan and the Telecommunications Advancement Foundation.

References

[1] J. Chen, H. Lu, and S. D. Bruda. "A Solution for Whitewashing in P2P Systems Based on Observation Preorder." *Proc. of International Conference on Networks Security, Wireless Communications and Trusted Computing (NSWCTC '09)*, pp.547–550, 2009.

[2] C. Costa and J. Almeida. "Reputation Systems for Fighting Pollution in Peer-to-Peer File Sharing Systems." *Proc. of the 7th IEEE International Conference on Peer-to-Peer Computing (P2P 2007)*, pp.53–60, 2007.

[3] Y.-M. Liu, S.-B. Yang, L.-T. Guo, W.-M. Chen, and L.M. Guo. "A Distributed Trust-based Reputation Model in P2P System." *Proc. of the 8h ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, pp.294–299, 2007.

[4] Y. Liu, W. Xue; K. Li, Z. Chi, G. Min, and W. Qu. "DHTrust: A Robust and Distributed Reputation System for Trusted Peer-to-Peer Networks." *Proc. of GLOBECOM 2010*, pp.1–6, 2010.

[5] S. Marti and H. Garcia-Molina. "Limited reputation sharing in P2P systems." *Proc. of the 5th ACM Conference on Electronic Commerce (EC '04)*, pp.91–101, 2004.

[6] Z. Malik and A. Bouguettaya. "Reputation Bootstrapping for Trust Establishment among Web Services." *Internet Computing, IEEE* , IEEE, 13(1): 40–47, 2009.

[7] A. P. de Pinninck, W. M. Schorlemmer, C. Sierra, and S. Cranefield. "A social-network defence against whitewashing." *Proc. of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, pp.1563–1564, 2010.

[8] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. "A scalable content-addressable network." *Proc. of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp.161–172, 2001.

[9] I. Reitzenstein and R. Peters. "Assessing Robustness of Reputation Systems Regarding Interdependent Manipulations." *E-Commerce and Web Technologies*, Lecture Notes in Computer Science, 2009, Vol. 5692, pp.288–299, 2009.

[10] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications." *ACM SIGCOMM Computer Communication Review*, 31(4): 149–160, 2001.

[11] Y.-M. Tseng and F.-G. Chen. "A free-rider aware reputation system for peer-to-peer file-sharing networks." *Expert Syst. Appl.*, 38(3): 2432–2440. 2011.

[12] Z. Xu, Y. He, and L. Deng. "A Multilevel Reputation System for Peer-to-Peer Networks." *Proc. of the 6th International Conference on Grid and Cooperative Computing (GCC 2007)*, pp.67–74, 2007.

[13] M. Yang, Y. Dai, and X. Li. "Bring Reputation System to Social Network in the Maze P2P File-Sharing System." *Proc. of the International Symposium on Collaborative Technologies and Systems (CTS 2006)*, pp.393–400, 2006.

[14] Y. Zhang and Y. Fang. "A Fine-Grained Reputation System for Reliable Service Selection in Peer-to-Peer Networks." *IEEE Transactions on Parallel and Distributed Systems*, 18(8): 1134–1145, 2007.

[15] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing." *Technical Report*, CSD-01-1141. University of California at Berkeley, 2001.

Distributed Real-Time Environment on Responsive Link

Hiroyuki Chishiro and Nobuyuki Yamasaki
Department of Information and Computer Science
Keio University, Yokohama, Japan
{chishiro,yamasaki}@ny.ics.keio.ac.jp

Abstract— *Responsive Link is a communication standard as specified in ISO/IEC 24740:2008 for distributed real-time environments. Responsive Link has multiple features to meet both hard and soft real-time requirements. Unfortunately, current real-time operating systems do not support the main features of Responsive Link. In addition, distributed real-time systems in noisy environments require revealing the tolerance of Responsive Link to achieve correct communication. We present APIs for Responsive Link in RT-Est real-time operating system. Using the APIs, users can make use of the main features of Responsive Link easily. Experimental evaluations perform the voltage tolerance test against noise and reveal the tolerance of Responsive Link.*

Keywords: Responsive Link, Responsive Multithreaded Processor, Distributed Systems, Real-Time Communication, Tolerance

1. Introduction

Robots [14], [18], [2] in distributed real-time systems have many nodes and require various topologies. In distributed real-time environments such as nuclear power plants, deep sea and outer space, there is much noise to interrupt correct communication. In addition, packets must be arrived by their deadlines in such noisy environments. However, existing communication standards have various limitations such as real-time requirement, topology and tolerance.

For example, wireless communication standards such as WirelessHART [7] reduce tolerance compared to wired communication standards. On the other hand, wired communication standards such as CAN [1] and FlexRay [6] suffer from the limitations of topologies. Since there are various shapes of robots, we require a topology-free communication standard. Also, these communication standards have priority inversion problems [17] because higher priority packets cannot overtake lower priority packets. Due to priority inversion problems, higher priority packets may miss their deadlines. In order to overcome the weakness of these communication standards, we have developed Responsive Link [20].

Responsive Link is a communication standard as specified in ISO/IEC 24740:2008 for distributed real-time environments. Responsive Link has multiple features to meet both hard and soft real-time requirements. For example, Responsive Link has two communication links: event link and data link. An event link is used to transmit packets with hard real-time requirements. On the other hand, a data link is used to transmit packets with soft real-time requirements. By the separate transmission of the event link and the data link, Responsive Link is easier to support hard and soft real-time communication than other communication standards. Unfortunately, current real-time operating systems do not support the main features of Responsive Link. In addition, distributed real-time systems in noisy environments require revealing the tolerance of Responsive Link to achieve correct communication.

We present APIs for Responsive Link in RT-Est real-time operating system [5]. Using the APIs, users can make use of the main features of Responsive Link easily. Experimental evaluations perform the voltage tolerance test against noise and reveal the tolerance of Responsive link.

The contribution of this paper is to implement APIs for Responsive Link and to build the experimental environment for tolerance. We believe that the APIs for Responsive Link help users to build distributed real-time systems easily and the method to evaluate the tolerance of communication standards is widely used.

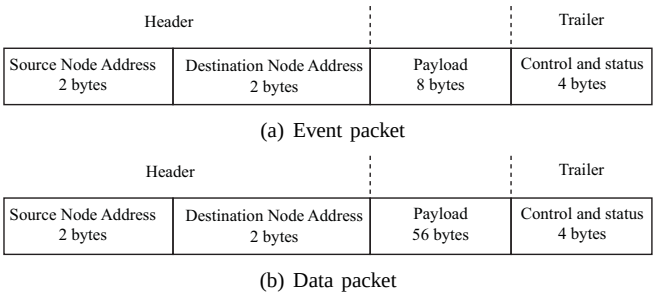
The remainder of this paper is organized as follows: Section 2 introduces the detail of Responsive Link. Section 3 presents the implementation of the APIs for Responsive Link. The effectiveness of Responsive Link is evaluated in Section 4. Section 5 compares our work with related work in distributed real-time environments. Finally we offer concluding remarks in Section 6.

2. Responsive Link

In this section, we introduce the detail of main features in Responsive Link [20].

2.1 Packet Overtaking

Real-time systems require real-time scheduling, which guarantees completing real-time tasks by their deadlines. In real-time scheduling, higher priority tasks can preempt lower priority tasks. In order to achieve this preemption in real-time communication, higher priority packets require overtaking lower priority packets in each node. Therefore, each packet in Responsive Link has a priority. If packets have the same priority, then the round-robin rule is applied. By the technique of the packet overtaking, real-time scheduling such as fixed-priority scheduling [13] and semi-fixed-priority scheduling [3], [4] can be adapted to packet scheduling.



(a) Event packet

(b) Data packet

Fig. 1: Format of packet

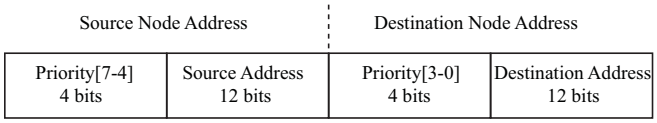


Fig. 2: Format of header

2.2 Format of Packet

Responsive Link supports two kinds of packets: event packet and data packet. Figure 1 shows the formats of the event packet and the data packet. An event packet is 16-byte length consisted of a 4-byte header, an 8-byte payload and a 4-byte trailer. The main use of the event packet is to transmit control commands and operations with hard real-time requirements. A data packet is 64-byte length consisted of a 4-byte header, a 56-byte payload and a 4-byte trailer. The main use of the data packet is to transmit image and sound data with soft real-time requirements.

Figure 2 shows the format of the header. A packet has a header that includes a 16-bit source node address and a 16-bit destination node address. The source node address includes a 4-bit priority[7-4] and a 12-bit source address. On the other hand, the destination node address includes a 4-bit priority[3-0] and a 12-bit destination address. The priority[7-4] has the 7-4 bits of the 8-bit priority and the priority[3-0] has the 3-0 bits of the 8-bit priority. Larger values have higher priorities in packets. The highest priority is 255 (0xff) and the lowest priority is 0 (0x00).

Figure 3 shows the format of the trailer. The trailer includes the following parameters.

- User Defined (UD): sets this bit to the user defined data.
- Full: sets this bit to 1 if all payload data are valid (data packet: 56 bytes, event packet: 8 bytes). Otherwise, set this bit to 0.
- Data Length: indicates the length of the valid payload data (data packet: 0-56, event packet: 0-8).
- Dirty[0-15]: indicates which word (4 bytes) in the packet has an error in case of a data packet or which byte in the packet has an error in case of an event packet. For example, if the 3^{rd} word of the data packet has an error, then set the Dirty2 bit to 1. If the 4^{th} word of the event packet has an error, then set the Dirty3 bit to 1.
- Start: sets this bit to 1 if this packet is the start packet. Otherwise, set this bit to 0.
- End: sets this bit to 1 if this packet is the end packet. Otherwise, set this bit to 0.
- Interrupt (Int.): generates an interrupt when this packet arrives the destination node if this bit is set to 1. Otherwise, set this bit to 0.
- Fatal: sets this bit to 1 by hardware if one byte in the packet has an unrecoverable fatal error. Otherwise, set this bit to 0.
- Correct: sets this bit to 1 by hardware if this packet has an error that has been corrected. Otherwise, set this bit to 0.
- Serial Number: indicates the serial number. The start packet has the serial number 0 and the serial number is incremented in the following packets. The serial number returns 0 after it arrives 7 and the sequence is repeated.

Byte	0	1	2	3	4	5	6	7
0	UD	Full	Data Length					
1	Dirty0	Dirty1	Dirty2	Dirty3	Dirty4	Dirty5	Dirty6	Dirty7
2	Dirty8	Dirty9	Dirty10	Dirty11	Dirty12	Dirty13	Dirty14	Dirty15
3	Start	End	Int.	Fatal	Correct	Serial Number		

Fig. 3: Format of trailer

2.3 Routing

Responsive Link achieves an end-to-end connection by setting the routing tables of all nodes along the transmission path from a source node to a destination node.

Figure 4 shows the routing table of Responsive Link. Each node has a routing table to control the route of the packet and the priority exchange function. The 32-bit reference part is the same as the header of the packet. On the other hand, the 16-bit referent part includes the following parts.

- Event Enable (EE): sets this bit to 1 if the event link is valid. Otherwise, set this bit to 0.
- Data Enable (DE): sets this bit to 1 if the data link is valid. Otherwise, set this bit to 0.
- Priority Exchange (PE): indicates whether the priority exchange function is valid.
- Priority[7-0] (P[7-0]): includes the new priority level (8 bits) which is valid if PE is set to 1.
- Link[4-0] (L[4-0]): indicates the output link numbers. The L[4-1] bits indicate the output link numbers of four physical ports in Responsive Link. On the other hand, the L[0] bit indicates the output link number to Dual Port Memory (DPM) in a processor. If multiple bits are

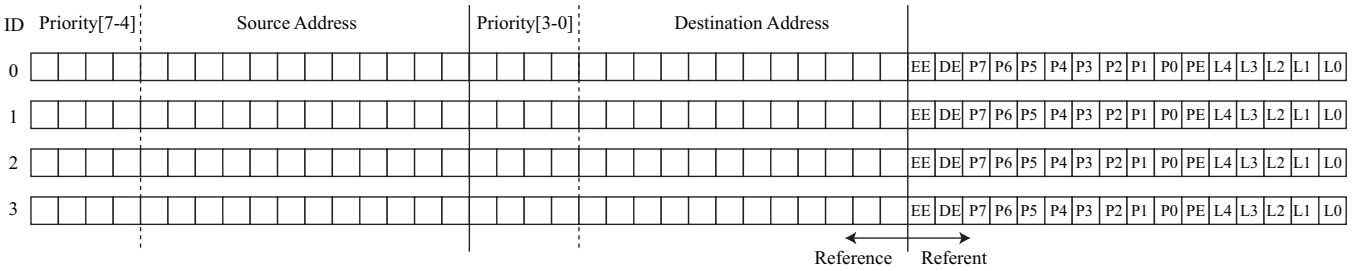


Fig. 4: Routing table

set to 1, then multicast is indicated. If all bits are set to 1, then broadcast is indicated.

Since the EE bit and the DE bit can be set independently, the event link and the data link may have different routes to the same destination.

Responsive Link supports the layer 1 to the layer 4 in OSI reference model. Using Responsive Link, we can implement communication protocols such as TCP with low overhead, compared to software implementation using other communication standards.

Unfortunately, current real-time operating systems do not support the main features of Responsive Link. Therefore, the APIs for Responsive Link are required to build distributed real-time systems easily.

3. Implementation

We implement APIs for Responsive Link in RT-Est real-time operating system [5]. Using the APIs, users can make use of the main features of Responsive Link easily.

3.1 Initialization

We implement the device driver of Responsive Link in `init_resplink` function. This function initializes the following parameters.

- SDRAM: sets the size of SDRAM for the packet overtaking by the priority of each packet selected with [None,8MB,16MB,32MB,64MB,128MB,256MB].
- Speed: sets the speed of Responsive Link selected with [50Mbps,100Mbps,200Mbps,400Mbps,800Mbps].
- Switch: initializes the switch, the encoder and the decoder of Responsive Link. The switch of Responsive Link supports the following modes.
  - Cut Through Mode: The switch starts to forward a packet before the whole packet has been received, as soon as the destination address is processed. This technique has the advantage of reducing latency and the disadvantage of decreasing tolerance.
  - Store and Forward Mode: The switch starts to forward a packet to a node where the packet is kept and sent at a later time to the destination node or another intermediate node.
- Interrupt: clears all interrupts of both the event link and the data link and enables a proper interrupt by users. For example, Responsive Link supports the following End Of Packet (EOP) interrupts.
  - Down: The cable of Responsive Link is unplugged.
  - Wakeup: The cable of Responsive Link is plugged.
  - Fatal: A fatal error occurs as described in Subsection 2.2.
  - Data-Out EOP: occurs if sending data packets in the range of the specified DPM.
  - Data-In EOP: occurs if receiving data packets in the range of the specified DPM.
  - Data Packet-In: occurs if receiving data packets, which enable their interrupt bits.
  - Event-Out EOP: occurs if sending event packets in the range of the specified DPM.
  - Event-In EOP: occurs if receiving event packets in the range of the specified DPM.
  - Event Packet-In: occurs if receiving event packets, which enable their interrupt bits.
- Link[4-0] (L[4-0]): indicates the output link numbers, as described in Subsection 2.3.
- Serial/Parallel[4-0]: indicates serial/parallel connection in each output link number. If an output link is parallel connection, set this bit to 1. If an output link is serial connection, set this bit to 0.
- DPM: initialize event in/out and data in/out registers.
  - Transmission Mode: If the transmission mode is Mode0, set this bit to 0. In Mode0, each packet includes each header and trailer. If transmission mode is Mode1, set this bit to 1. In Mode1, first of all, Responsive Link transmits the payload of each packet at the head of the DPM sequentially. After the payload of each packet, Responsive Link next transmits the header and the trailer of each packet sequentially. Therefore, all packets arrive at the same destination node address.
  - Interrupt: If this bit is set to 1, an EOP interrupt occurs.
  - Dreq: If this bit is set to 1, perform DMA transfer of the packet by the DMA counter.
  - From_Addr/To_Addr: These control registers have

```
struct resplink_header {
  unsigned int high_priority:4;
  unsigned int src_addr:12;
  unsigned int low_priority:4;
  unsigned int dst_addr:12;
} __attribute__((packed));
```

Fig. 5: struct resplink_header

```
struct eventlink_packet {
  union {
    uint32_t addr;
    struct resplink_header header;
  } u;
  uint8_t payload[8];
  uint32_t trailer;
};
```

Fig. 6: struct event_packet

```
struct datalink_packet {
  union {
    uint32_t addr;
    struct resplink_header header;
  } u;
  uint8_t payload[56];
  uint32_t trailer;
};
```

Fig. 7: struct data_packet

```
void send_eventlink(unsigned int mode,
  struct event_packet *ep);
void send_datalink(unsigned int mode,
  struct data_packet *dp);
int recv_eventlink(uint8_t *buf);
int recv_datalink(uint8_t *buf);
```

Fig. 8: send/receive functions

same features. For ease of comprehension to users, the name of each control register is different.

Users can configure the above parameters for target distributed real-time applications. Therefore, Responsive Link can support more various types of distributed real-time applications than other communication standards.

3.2 Packet

We explain the format of the event packet and the data packet.

Figure 5 shows struct resplink_header. As shown in Figure 2, struct resplink_header includes a 4-bit high_priority, a 12-bit src_addr, a 4-bit low_priority and a 12-bit dst_addr. In addition, struct resplink_header has the packed type attribute.

Figure 6 shows struct event_packet. As shown in Figure 1(a), struct event_packet has a 4-byte header, a 8-byte payload and a 4-byte trailer. In struct eventlink_packet, union u has a 4-byte address and struct resplink_header. For example, this 4-byte address is used to write header to the reference part in routing tables and this header is used to set the priority, the source address and the destination address.

Figure 7 shows struct data_packet. As shown in Figure 1(b), struct data_packet has a 4-byte header, a 56-byte payload and a 4-byte trailer. Like struct event_packet in Figure 6, struct data_packet also has union u.

Figure 8 shows send/receive functions in event link and data link. The first argument of send_eventlink and send_datalink functions is mode, which selects Mode0 or Mode1 if the first argument is 0 or 1 respectively. The second argument of send_eventlink is the pointer of struct event_packet. Also, the second argument of send_datalink is the pointer of struct data_packet. The first argument of recv_eventlink and recv_datalink functions is the pointer of the 1-byte character. This pointer indicates the head of the buffer receiving event packets and data packets respectively.

3.3 Routing Table

Figure 9 shows add/delete functions to add/delete routing tables respectively. add_rtable function adds a rule to both reference and referent parts in the routing table, as shown in Figure 4. Then the rule is set to the unique identifier. On the other hand, delete_rtable function deletes the rule selected by id.

Note that if registering the same rules of routing tables, the rule with the smallest id in the same rules is applied to routing packets.

3.4 Example

In this subsection, we explain an example of usage with APIs for Responsive Link.

Figure 10 shows an example of sending/receiving data packets in usermain function. First of all, struct datalink_packet sets the header, the payload and the trailer of the data packet. The priority of the header is 0x55, the source address is 0xa5a and the destination address is 0xa5a. Since this is an example of the loopback transmission, the source address is the same as the destination address. The trailer of the data packet sets Full bit to 1 because all payload data are valid.

Next we set new_priority to 0x11. Then init_resplink function is called to initialize the parameters of Responsive Link, as described in Subsection 3.1. Now we call add_rtable function to set the routing table. The first call of add_rtable function sets the priority, the source address and the destination address of dp. The priority exchange function is valid and changes the priority of data packet from 0x55 to 0x11. The output link of the data packet is L[1]. The second call of add_rtable

```
void add_rtable(uint32_t reference,
                uint16_t referent);
void delete_rtable(unsigned int id);
```

Fig. 9: add/delete functions

Table 1: Specification of RMTP

Clock frequency	31.25MHz
SDRAM	64MB
SRAM	256KB
I-Cache/D-Cache	each 32KB (Harvard)
Fetch width	8
Issue width	4
Integer register	32-bit × 32-entry × 8-set
Integer renaming register	32-bit × 64-entry
FP register	64-bit × 8-entry × 8-set
FP renaming register	64-bit × 64-entry
ALU	4 + 1 (Divider)
FPU	2 + 1 (Divider)
64-bit ALU	1
FP vector unit	1 (4FPU × 2line)
Branch unit	2
Memory access unit	1

function sets `new_priority`, the value of which is 0xff. The source address and the destination address are set to those of `dp`. The priority exchange function is also valid and changes the priority of data packet from 0x11 to 0xff. The output link of the data packet is L[0]. We call `send_datalink` function to send the data packet with Mode0. After sending the data packet, the program calls `recv_datalink` function repeatedly until receiving the data packet. The return value of `recv_datalink` function is the length of received data. If receiving the data packet, the busy loop is finished.

Note that the priority of the data packet is finally set to 0xff so that the header of the data packet is changed from 0x5a5a5a5a to 0xfa5afa5a. If the second call of `add_rtable` sets `new_priority`, the value of which is 0x55, the packet is not transmitted to L[0] because the original priority of the data packet is 0x55. As a result, the header of the data packet is changed to 0x5a5a5a5a and the data packet is transmitted to L[1] repeatedly.

```
void usermain(void)
{
  unsigned int cur_priority, src, dst,
    new_priority;
  /* initialize data packet */
  struct datalink_packet dp = {
    .u.header = {
       .high_priority = 0x5,
       .src_addr = 0xa5a,
       .low_priority = 0x5,
       .dst_addr = 0xa5a,
    },
    .payload = {0},
    .trailer = 0x40000000,
  };
  uint8_t buf[0x40];
  new_priority = 0x11;

  /* initialize Responsive Link */
  init_resplink();

  /* add rule to routing table */
  add_rtable(dp.u.addr,
             (0x3 << 14) |
             (new_priority << 6) |
             (0x1 << 5) | 0x1);
  cur_priority = new_priority;
  new_priority = 0xff;

  /* add rule to routing table */
  add_rtable(((cur_priority & 0xf0) << 24) |
              (dp.u.addr & 0x0fff0fff) |
              ((cur_priority & 0xf) << 12),
              (0x3 << 14) |
              (new_priority << 6) |
              (0x1 << 5) | 0x0);

  /* send data packet */
  send_datalink(0, &dp);

  /* wait until receiving packet */
  while (recv_datalink(buf) == 0)
    ;
}
```

Fig. 10: Example of sending/receiving data packets

4. Experimental Evaluations

4.1 Experimental Environments

The experimental evaluations use Responsive Multi-threaded Processor (RMTP) [19] which has prioritized SMT architecture with MIPS and RMTP-specific instructions. Table 1 shows the specification of RMTP. We implement APIs for Responsive Link [20] in RT-Est real-time operating system [5] on RMTP. In order to transform RMTP-specific instructions to machine language, we have developed a cross-compiler for RMTP, which extends `gcc` version 3.4.3 for MIPS.

Figure 11 shows our experimental environment. There are two RMTPs, which are connected by Responsive Link. The noise generator covers the cable of Responsive Link. The baud rate of Responsive Link is selected within [50Mbps,100Mbps,200Mbps,400Mbps,800Mbps]. In `init_resplink` function, the switch mode is set to the store and forward mode because the store and forward mode has higher tolerance than the cut through mode. In addition, the transmission mode is set to Mode0 and the connection method is serial. We use the data link in experimental evaluations.

We assume the actual noise in noisy environments where a humanoid robot [14] runs. Table 2 shows the specification of the noise wave.

In order to evaluate the tolerance of Responsive Link, we perform the voltage tolerance test. First of all, we set the

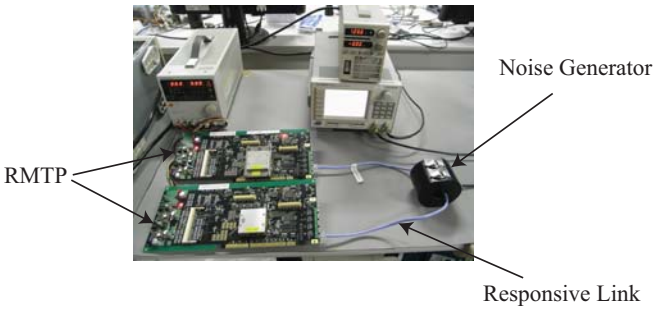


Fig. 11: Experimental environment

Table 2: Specification of noise wave

Frequency	12MHz
# of bursts	5
Wave shape	Triangle

voltage of the noise wave to 1.0V in the noise generator. Next, we check whether sending/receiving data packets in the noisy connection. If Responsive Link can send/receive data packets, we raise the voltage of the noise wave by 0.1V. If Responsive Link cannot send/receive packets due to noise, the value subtracting 0.1V from the current voltage is that of the voltage tolerance against noise in this environment. We perform this experiment by 10 times in each baud rate and measure the minimum, average and maximum values of the voltage tolerance.

4.2 Overhead

Table 3 shows the overhead of APIs for Responsive Link. `init_resplink` function sets the various parameters of Responsive Link. As a result, this function has the most cycles in evaluated functions. `add_rtable` function has low overhead because this function only writes a 32-bit reference part and a 16-bit referent part and sets the identifier of the rule which is the smallest in empty ids. `send_datalink` function also has low overhead because this function sets the control registers of the data link and the DMAC. We measure the overhead of `recv_datalink` function from entering to exiting this function. `recv_datalink` function checks whether receiving data packets. If receiving data packets, write the data packets to the specified memory. Therefore, `init_resplink` function has higher overhead than `add_rtable` and `send_datalink` functions.

4.3 Voltage Tolerance Test

Figure 12 shows the result of the voltage tolerance in the loopback transmission. If the baud rate of Responsive Link is faster and faster, the voltage tolerance is usually lower and lower.

Figure 13 shows the result of the voltage tolerance in the point-to-point transmission. Each voltage tolerance in the point-to-point transmission has slightly higher voltage

Table 3: Overhead of APIs

function	# of cycles
`init_resplink`	1,681
`add_rtable`	91
`send_datalink`	27
`recv_datalink`	344

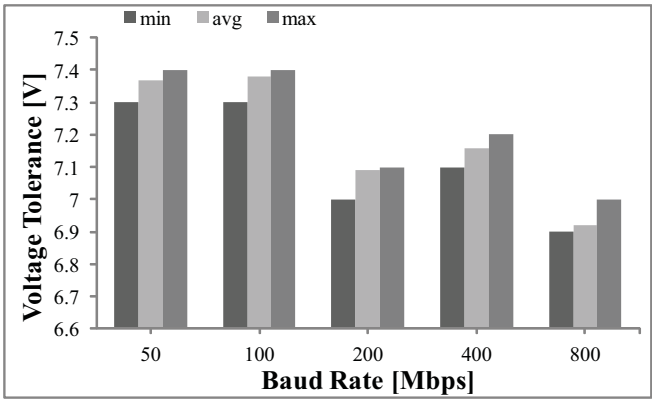


Fig. 12: Loopback transmission

tolerance than that in the loopback transmission in Figure 12. That is to say, the point-to-point transmission improves the tolerance compared to the loopback transmission. Like the loopback transmission, if the baud rate of Responsive Link is faster and faster, the voltage tolerance is also usually lower and lower.

5. Related Work

We compare our work with other work in distributed real-time environments.

TAO [16] is a middleware, which is compliant with RT-CORBA [8]. In addition, TAO implements IIOP protocol, which is an TCP/IP implementation of GIOP protocol specified in CORBA [9], [10], [11] by Ethernet. However, it is difficult to avoid packet collision by Ethernet. In contrast, Responsive Link supports layer 1 to layer 4 in OSI reference model and has four physical ports to avoid packet collision.

Except Ethernet, CAN [1] and FlexRay [6] are also wired communication standards. CAN supports only bus topology and FlexRay supports both bus and star topologies. Using these standards, a message packing algorithm is proposed [12]. Distributed control robots require various topologies so that topology-free communication standards such as Responsive Link have more compatibility than CAN and FlexRay.

In wireless communication standards, real-time packet scheduling over WirelessHART [7] is proposed [15]. Like Responsive Link, wireless communication standards are also topology-free. In general, wireless communication standards frequently occur packet collision/loss compared to wired communication standards so that the analysis of the worst case arrival time in each packet is difficult. Therefore,

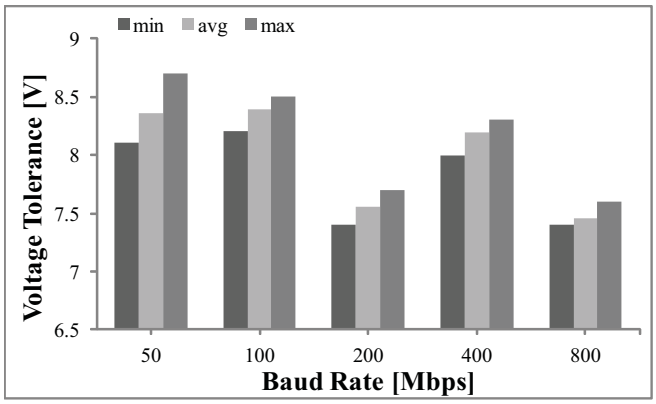


Fig. 13: Point-to-point transmission

Responsive Link is better than wireless communication standards in distributed real-time environments.

6. Concluding Remarks

We presented the APIs for Responsive Link in RT-Est real-time operating system. In particular, we explain how to use the APIs for data link. Using the APIs, users can build distributed real-time systems easily. In addition, we introduce a method to evaluate the tolerance of communication standards. Experimental evaluations reveal the voltage tolerance against noise in Responsive Link. Using the results, users can select a proper baud rate to communicate between nodes correctly in noisy environments.

In future work, we will implement the APIs for packet scheduling with fixed-priority scheduling [13] and semi-fixed-priority scheduling [3], [4] over Responsive Link. The analysis of the worst case arrival time in each packet is intriguing.

Acknowledgement

This research was supported in part by CREST, JST. This research was also supported in part by Grant in Aid for the Global Center of Excellence Program for "Center for Education and Research of Symbiotic, Safe and Secure System Design" from the Ministry of Education, Culture, Sport, and Technology in Japan. This research was also supported in part by Grant in Aid for the JSPS Fellows.

References

[1] I. 11898. Road vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication, 1993.

[2] H. S. Ahn, Y. M. Beak, I.-K. Sa, W. S. Kang, J. H. Na, and J. Y. Choi. Design of Reconfigurable Heterogeneous Modular Architecture for Service Robots. In *IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems*, pages 1313–1318, Sept. 2008.

[3] H. Chishiro, A. Takeda, K. Funaoka, and N. Yamasaki. Semi-Fixed-Priority Scheduling: New Priority Assignment Policy for Practical Imprecise Computation. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 339–348, Aug. 2010.

[4] H. Chishiro and N. Yamasaki. Global Semi-fixed-priority Scheduling on Multiprocessors. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 218–223, Aug. 2011.

[5] H. Chishiro and N. Yamasaki. RT-Est: Real-Time Operating System for Semi-Fixed-Priority Scheduling Algorithms. In *Proceedings of the 2011 International Symposium on Embedded and Pervasive Systems*, pages 358–365, Oct. 2011.

[6] F. Consortium. FlexRay communications system - protocol specification v3.0. `http://www.flexray.com/`, Dec. 2009.

[7] H. C. Foundation. `http://www.hartcomm.org/`.

[8] O. M. Group. *Real-time CORBA Specification*, formal/05-01-04 edition, Jan. 2005.

[9] O. M. Group. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 Part 1: CORBA Interfaces*, formal/08-01-04 edition, Jan. 2008.

[10] O. M. Group. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 Part 2: CORBA Interoperability*, formal/08-01-07 edition, Jan. 2008.

[11] O. M. Group. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 Part 3: CORBA Component Model*, formal/08-01-08 edition, Jan. 2008.

[12] T. Ishigooka and F. Narisawa. Message Packing Algorithm for CAN-Based Legacy Control Systems Mixed with CAN and FlexRay. *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, 3(1):88–97, 2010.

[13] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20:46–61, 1973.

[14] I. Mizuuchi, Y. Nakanishi, Y. Sodeyama, Y. Namiki, T. Nishino, N. Muramatsu, J. Urata, K. Hongo, T. Yoshikai, and M. Inaba. Advanced Musculoskeletal Humanoid Kojiro. In *Proceedings of the 2007 IEEE-RAS International Conference on Humanoid Robots*, pages 294–299, 2007.

[15] A. Saifullah, Y. Xu, C. Lu, and Y. Chen. Real-Time Scheduling for WirelessHART Networks. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 150–159, Nov. 2010.

[16] D. C. Schmidt, B. Natarajan, A. G. N. Wang, and C. Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

[17] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[18] T. Taira, N. Kamata, and N. Yamasaki. Design and Implementation of Reconfigurable Modular Robot Architecture. In *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3566–3571, 2005.

[19] N. Yamasaki. Responsive Multithreaded Processor for Distributed Real-Time Systems. *Journal of Robotics and Mechatronics*, 17(2):130–141, 2005.

[20] N. Yamasaki. Responsive Link for Distributed Real-Time Processing. In *Proceedings of the 10th International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 20–29, Jan. 2007.

Modeling Packet Processing Time in a Multiprocessor Network Traffic Monitoring System

Luis Zabala, Armando Ferro, and Alberto Pineda
Networking, Quality and Security (NQaS) Research Group
University of the Basque Country (UPV/EHU), Bilbao, Spain

Abstract - *Nowadays traffic monitoring is a must for many purposes such as QoS monitoring, IDS, antivirus, network problem detection and so on. Deployment of high speed networks implies problems with this kind of systems to be able to cope with all the traffic in the network. Therefore, it would be interesting to know in advance whether a traffic monitoring system will be able to do its task correctly, or it needs more processing power. This paper presents a mathematical model for multiprocessor traffic monitoring systems that use commodity hardware and general purpose operating systems. In order to establish the different elements of the model we have identified the different stages of the trip of packets from wire to application, as well as the particular behavior of the system and the computational cost for each one of them. With this information, we have built up a model based on a closed queuing network that simulates these different stages of the monitoring system and the possible packet losses. This model allows us to estimate the performance of the monitoring application in terms of throughput.*

Keywords: *traffic monitoring; performance evaluation; closed queuing network; multi-server system*

1 Introduction

The performance of network traffic monitoring systems has a great importance in all kind of network devices. Moreover, there are some applications related to network security and packet analysis which are particularly sensitive to network capturing performance. These include IDS (Intrusion Detection Systems), antivirus, QoS (Quality of Service) analysis systems. In order to achieve Gigabit-order analysis capabilities, most of these systems are forced to use specific hardware and operating systems. This results in more expensive devices and longer development processes.

On the other hand, traditionally, the performance degradation of those systems based on commodity hardware and general purpose operating systems is mostly caused by a design focused on network applications, not on extensive network packets capturing [1]. However, last investigations have demonstrated that packet capture has been improved by enhancing general purpose operating systems for traffic analysis. These results are encouraging because today's commodity hardware offers features and performance that just a few years ago were only provided by costly custom hardware design [2] [3]. Even then, there are still deficiencies to adapt to new architectures such as multithreaded and multi-cored ones.

In order to continue the improvement of this second kind of systems, it is necessary to identify the design flaws that have an impact on capturing performance. So, it is also important to develop theoretic models and simulation tools that help us in the design and development cycles.

The experience of our research group in the development of a traffic capture and analysis system has encouraged our study of the packet capturing subsystem in general purpose operating systems, especially Linux and its effect on the overall packet analysis performance. In order to improve the traffic monitoring systems' performance we need data about where the packets are lost and why. These data allow us to predict the performance of our system without running real simulations and, therefore, without designing and implementing several prototypes. So, in order to help us in the design and cut down development cycles we have built a model of the different stages of the packet capturing subsystem of Linux Kernel. After obtaining real measurements of the computational costs of each stage we have developed a model that provides us with data about the packet capturing and analysis capabilities, such as the number of packet lost in each stage.

The paper is organized as follows. The background of our work is presented in Section 2. Then, Section 3 describes the analytical model of the packet processing consumptions. The model validation is explained in Section 4. Finally, we conclude the paper in Section 5.

2 Background

In the previous work [4], our research group proposed a prototype of a traffic monitoring system for high speed networks called Ksensor. It works at kernel-level and it is based on Linux. The experimental study of Ksensor has brought some aspects that are worth analyzing.

2.1 Problem Formulation

The aspects mentioned before are related to the time the system takes in order to capture or analyze a packet. We have noticed that this time depends on the network traffic rate and the analysis load which is managed by the traffic monitoring

system. In the laboratory, in order to test Ksensor, we simulate different analysis loads implementing four loops that execute 0, 1000, 5000 and 25000 cycles respectively.

Fig. 1 shows the values of a parameter measured experimentally on Ksensor. In particular, Fig. 1 presents the average softIRQ time per captured packet. This parameter belongs to the capturing stage and, as can be observed, it varies with the injection rate and the analysis loads. It is worth mentioning the three areas that we can observe on the 1 Kcycle analysis load curve. Moreover, in the graphic it can be seen that it takes more time capturing a packet with 0 Kcycle analysis load (null) than with 25 Kcycle analysis load. On the contrary, the average values of the analysis time per packet are similar for all the range of injection traffic rates.

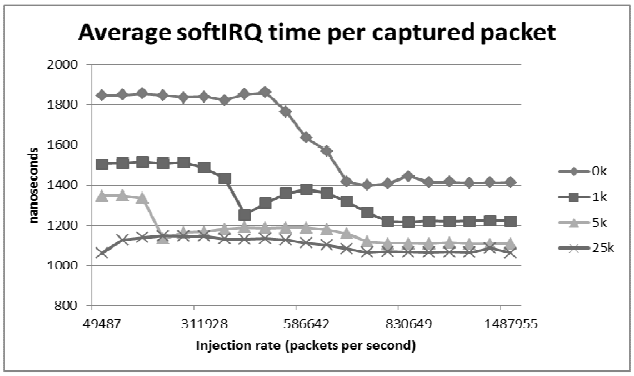


Figure 1. Average softIRQ per captured packet.

Thus, we intend to make a proposal of a mathematical model which considers the capturing and the analysis stages with their possible variability of CPU consumptions. Certain aspects correspond to our group's own development. However, we consider that these characteristics can also be extrapolated to more general systems.

Our initial hypotheses take us to consider some packet losses along the capturing stage. Hence, as it happens when duplicate packets or IP fragmentation losses are detected; some activities entail a waste of CPU. In the case that concerns us, due to the shortage of resources or, even, due to control policies of the monitoring application, some packets can be discarded in the capturing stage, producing a CPU consumption that does not result in a captured packet.

2.2 Generic Traffic Monitoring Systems

If we consider a typical traffic monitoring system, we usually come across three different phases: capture phase, basic analysis or filtering phase and complex analysis phase. The capture phase can be divided into: hardware processing stage (network interface card), driver processing stage and kernel processing stage.

The basic analysis phase is based on classifying each received packet after studying its features to determine whether the packet must be further analyzed or discarded. This task must be carried out for every captured packet. The real packet processing (e.g. intrusion detection algorithm, QoS algorithm, etc.) is applied in the complex analysis phase but only to those packets that successfully passed the basic analysis phase.

In the next section, we will focus on analyzing the parts of the Linux packet capture subsystem. Since the impact of each parameter depends on its computational costs, we work out the cost of every phase along the capture system and identify which stages would lose packets more likely.

2.3 Packet Losses

As it has been said before, we consider that there are packet losses in the capturing stage. Because of that, it is important to study the capturing system of Linux.

After a thoughtful analysis of the path followed by a packet through Linux operating system, we can identify the points where packet losses are more liable. As a result, these points must be considered in the design of packet capture model. Although Linux is a specific case, the path followed by packets in other operating systems is quite similar to this.

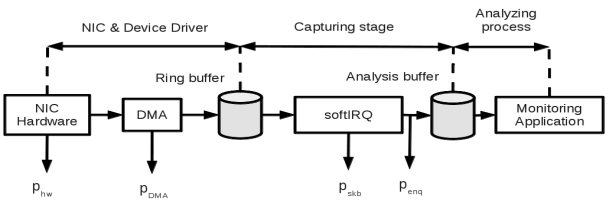


Figure 2. Packet losses in the packet receiving process with Linux networking subsystem

Fig. 2 shows the trip of a packet from its ingress into a Linux end system to its final delivery to the monitoring application, taking into account the possible packet losses along the journey. When a packet arrives to the NIC a hardware interruption (hardIRQ) is generated. The network interface card (NIC) captures packets and copies them in an internal buffer. Then, the DMA (Direct Memory Access) engine of the card, without CPU intervention, is in charge of moving these packets to a special allocation area in the main memory of the system called DMA area. The driver is in charge of attending the hardIRQ. When arrival rate is high, this interruption is generated for a bunch of packets. In order to extract the packets from the DMA area, the driver uses polling mechanism. After any interruption, a software interruption (softIRQ) is scheduled in order to complete the capture of packets. Packets are then moved, during the softIRQ, from the DMA area to another space in the main memory, creating a skb element list. Skb elements are buffers in which the kernel handles network packets [5]. Next, the softIRQ has to move the packet to the monitoring application's analysis buffer.

Packet losses due to capture deficiencies are represented by p_{hw} and are not very common. Packet losses due to DMA transfer errors are represented by p_{DMA}. A p_{DMA} will occur when packet arrival rate is very high and the NIC allocates in

main memory more packets than the system is able to manage, causing an overflow of the DMA's reserved space in main memory. When the packets are moved from the DMA area to the skb element list there can be eventual losses, which are also very uncommon. These losses are represented by p_{SKB}. Finally, p_{enq} represents the proportion of packets which cannot be enqueued in the monitoring application's analysis buffer. Obviously, the length of this queue is limited, so that an overflow of this queue becomes quite likely only at very high arrival rates.

3 Packet Process Consumptions Model

This section introduces an analytical model which works out, firstly, the different stages which have been identified in a multiprocessor traffic monitoring system and, secondly, the possible packet losses in the journey from the network card to the analysis application. For a large number of arrivals (heavy traffic conditions), the multiprocessor architecture can be modeled as a closed queuing network [6].

3.1 Description of the Model

The proposal for modeling packet process consumptions of a traffic monitoring system is showed in Fig. 3. It consists in a closed queuing network where computer consumptions are related to the service capacity of the queues. Two parts can be distinguished; the upper one has a set of multi-server queues which represents the traffic monitoring system with its different stages (capture, filtering and analysis), processing abilities (μ_{Ci}, μ_F, μ_A) and possible packet losses (p_{hw_dma}, p_{skb}, p_{enq}). The part on the left models the injection of network traffic with λ rate. This simple queue closes the network and the number of packets that are permitted is N. With heavy traffic conditions, the departure of a packet from the traffic monitoring system triggers the injection of an already waiting packet.

Four different stages have been distinguished for the closed network, each one with a specific function:

- Capture stage: it represents the functionalities provided by the operating system which is responsible for capturing packets and moving them from the NIC to the memory of the analysis application. It comprises treatments of device controllers and attention paid by kernel to interruptions (hardIRQ and softIRQ) due to packet arrival. This stage is divided into three multi-server queues with rates μ_{C1}, μ_{C2}, μ_{C3} (measured in packets per second) capacity, due to the need to differentiate the packet losses inside it. For this, p_{skb} is defined as the probability of having every ring buffer descriptor full and p_{enq} is defined as the probability of not enqueuing packets to the analysis buffer.
- Filtering stage: it is modeled by a multi-server queue with rate μ_F. According to some rules, captured packets are filtered and, finally, only selected packets are analyzed in the next stage. This stage represents the amount of time spent on this basic treatment.
- Analysis stage: it is integrated by a multi-server queue with rate μ_A. It simulates the complex analysis treatment that the system does to packets that need further analysis. As not all the packets need to be processed in this stage, a rate called q_a indicates the proportion of the received packets that has to be analyzed.
- Traffic injection stage: it is a simple queue of λ rate. This stage simulates the arrival of packets to the system with λ rate.

Since the number of packets in the closed network is fixed to N, the traffic injection queue can be empty. This situation simulates the blocking and new packets will not be introduced on the system. The model also considers the possibility of losing packets due to deficiencies at NIC or DMA's transfer errors with p_{hw_DMA} probability. However, since it is generally recognized that they are very uncommon, those losses will be negligible and $p_{hw\_DMA}=0$ for solving the model.

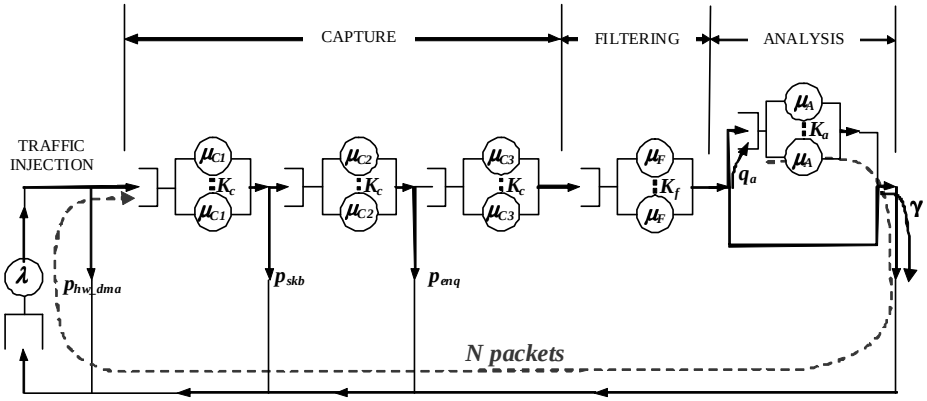


Figure 3. Model based on closed queuing network for packet process consumptions

We assume that the number of processors of the system is K and each stage can be served by a different number of processors, having K_c available processors for capturing, K_f for filtering and K_a for analysis being $K_c \leq K$, $K_f \leq K$ and $K_a \leq K$. So, the parallelizing level can be different in each phase. Another aspect to consider is that packets cannot flow freely in the closed network, because the sum of packets attended in the servers that represent the traffic monitoring system never exceeds the maximum number of processors available.

3.2 Simplifications of the Model

The model presented in Fig. 3 is very general, but some simplifications are possible. First, it is worth mentioning that both, the flowing traffic and the processing capacity at the nodes, are modeled by Poisson arrival rates and exponential service rates. Poisson's distributions are considered to be acceptable for certain types of network traffic, for instance, modeling the voice traffic, as explained in [7]. This assumption can be relaxed to more general processes such as MAPs (Markov Arrival Processes) [8] or non-homogeneous Poisson processes. However, for some other types of traffic such as Ethernet network, packet arrivals do not follow a Poisson process but are rather bursty [9]. The case we are dealing with is slightly different because the packet arrival is not directly the traffic of the Ethernet network, but it is the incoming packets from the network card's buffer to the kernel memory area via DMA. Regarding service rate modeling, although program's code has a quite deterministic behavior, some randomness is introduced by Poisson incoming traffic, variable length of packets and kernel scheduler uncertainty. An analytical solution can become unmanageable when considering non-Poisson arrivals, even presuming general service times. Despite of these limitations, we will keep working with these assumptions for simplicity of the analysis and, as will be demonstrated in Section 4, the results obtained from our models were closely matching to results obtained from real experimental measurements.

Apart from that, the main feasible simplification preserving the identity of the system is to replace the whole traffic monitoring system with its Norton equivalent [10]. Our theoretical model has exponential service rates in all stages, so applying the Norton equivalence, the new equivalent queue will have a state-dependent service rate $\mu_{eq\_TMS}(n)$.

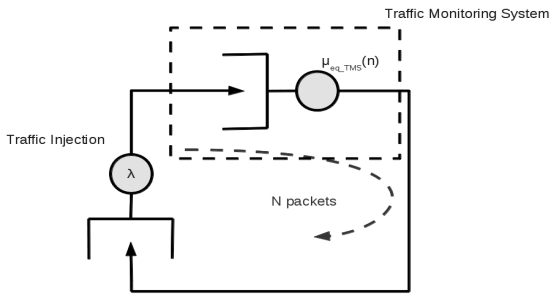


Figure 4. Simplified model with the Norton equivalent.

Therefore, with the aim to obtain the simplified model of Fig. 4, we apply the calculation of the Norton equivalent to the general model of Fig. 3 several times as we will see later.

In the study of this model, we observe that the same topology is repeated at different levels of abstraction. This topology corresponds with a closed network model with two multi-server queues where the output flow of the first queue goes to the second one with a probability of 1-p, whereas it comes back to the first one with a probability of p, as shown in Fig. 5. This structure usually occurs in every processing stage.

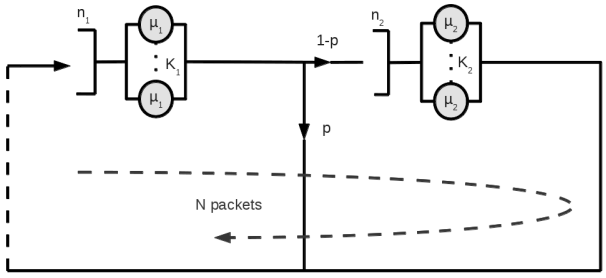


Figure 5. Topology repeated at different levels of abstraction.

3.3 Equations of the Repeated Topology

In order to get the Norton equivalent of the traffic monitoring system, first, we calculate the state probabilities for the repeated topology (see Fig. 5), putting N packets in circulation through the closed network, but assuming that the total system can have at most K packets being served and the rest waiting in the queue. We also take into account the limitation of K_c, K_f and K_a processors in each stage.

The state diagram for this topology is presented in Fig. 6. In this model we are representing the state i of the multi-server queue on the left of Fig. 5 and we will consider μ_i as the service capacity for the state i. N packets are flowing through the closed network and when there are i packets in the multi-server queue on the left, the rest, N-i, are in the multi-server queue on the right. The probability of that state i is represented as $p_N(i)$. Finally, the output of the multi-server queue with rate $\mu_2(n_2)$ is the input of the multi-server queue with rate $\mu_1(n_1)$.

It is possible to deduce the balance equations from the diagram of states and, subsequently, the expression of the probability of any state i as a function of the probability of state zero $p_N(0)$.

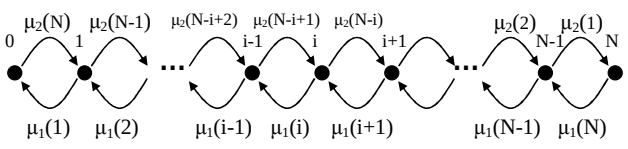


Figure 6. State diagram for the multiple queue with K_1 servers ($N \leq K$).

$$\left.\begin{array}{l} p_N(0)\cdot\dfrac{\mu_2(N)}{q}=p_N(1)\cdot\mu_1(1) \\ p_N(1)\cdot\dfrac{\mu_2(N-1)}{q}=p_N(2)\cdot\mu_1(2) \\ \ldots\ldots\ldots\ldots \\ p_N(N-1)\cdot\dfrac{\mu_2(1)}{q}=p_N(N)\cdot\mu_1(N) \end{array}\right\} \Rightarrow p_N(n+1)=\frac{\mu_2(N-n)}{q\cdot\mu_1(n+1)}\cdot p_N(n) \quad \forall n=1,\cdots,N \quad (1)$$

The term $p_N(n)$ indicates the probability of having n packets at queue 1 being N the number of packets in the closed network. For $N \leq K$ (being K the total number of processors):

$$p_N(n)=\frac{\prod_{i=N-n+1}^{N}\mu_2(i)}{q^n\cdot\prod_{i=1}^{n}\mu_1(i)}\cdot p_N(0) \quad \text{with } p_N(0)=\frac{1}{S_N}$$
$$S_n=1+\sum_{i=1}^{n}\frac{\prod_{j=n-i+1}^{n}\mu_2(j)}{q^i\cdot\prod_{j=1}^{i}\mu_1(j)} \quad 1\leq n\leq N \quad (2)$$

For every value of N, we calculate the throughput of the closed network and it will be the Norton equivalent which we want.

$$\mu_{eq}(N=1)=\sum_{i=1}^{N=1}\mu_1(i)\cdot p_1(i)=\mu_1(1)\cdot p_1(1)$$
$$\mu_{eq}(N=2)=\sum_{i=1}^{N=2}\mu_1(i)\cdot p_2(i)=\mu_1(1)\cdot p_2(1)+\mu_1(2)\cdot p_2(2) \quad (3)$$
...
$$\mu_{eq}(N)=\sum_{i=1}^{N}\mu_1(i)\cdot p_N(i)=\mu_1(1)\cdot p_N(1)+\mu_1(2)\cdot p_N(2)+..+\mu_1(N)\cdot p_N(N)$$

Taking into account these expressions, we can develop the equations of the whole model as will be detailed below.

3.4 Calculation of the Norton equivalent of the traffic monitoring system with losses

In order to calculate the Norton equivalence of the traffic monitoring system with losses, it is necessary to apply equations (1) and (2) several times.

First we compute the Norton equivalent for the filtering and the analysis stages. Then a second Norton equivalent is computed with the result of the first Norton equivalent and the multi-server queue with rate μ_{C3}. Later, a third Norton equivalent is computed with the result of the second Norton equivalent and the multi-server queue with rate μ_{C2}. Finally, a fourth Norton equivalent is computed with the result of the third Norton equivalent and the multi-server queue with rate μ_{C1}. This last result is the Norton equivalent of the traffic monitoring system. The service rate of the traffic monitoring system will be different for every value of N, i.e. it will be a state-dependent service rate.

For the calculation of the Norton equivalence, it must be remembered that the state diagram makes sense for values of N that are less or equal to the highest number of processors.

3.5 Solution for the Closed Network Model with Incoming Traffic

The previously explained Norton equivalence takes into consideration the internal problems of the traffic monitoring system related to the number of available processors. Now we complete the model adding the traffic injection queue to the equivalent system calculated before and the model of the entire system with incoming traffic corresponds with Fig. 4. Hence, the entire system under traffic load is modeled as a closed network with an upper queue, which is the Norton equivalent of the traffic monitoring system, and a simple queue on the left of the diagram, simulating the injection of network traffic with λ rate. In this closed network, a finite number N of packets circulates. This number N is greater than K, the number of available processors.

The analytical solution of this model is similar to that proposed for the repeated topology, taking into account the following: the arrival rate is λ and it is not state-dependent; the service rates μ_1, μ_2..., μ_p correspond with the calculation of the Norton equivalent of the traffic monitoring system, thus $\mu_n=\mu_{eq\_TMS}(n)$ with values of n from 1 to p; for states n with $K<n\leq N$, $\mu_n=\mu_K=\mu_{eq\_TMS}(K)$.

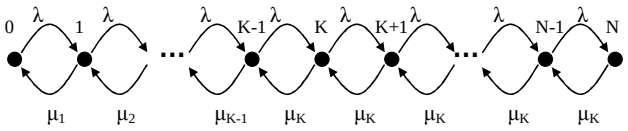


Figure 7. State diagram for the traffic monitoring system with incoming traffic (N>K).

Fig. 7 shows the related state diagram. This model allows us to calculate the theoretical throughput of the traffic monitoring system for different loads of network traffic.

$$\gamma=\lambda\cdot(1-p(N)) \quad (4)$$

4 Model Validation

This section explains the validation tests of the analytical model presented in this paper. The aim is to compare theoretical results with those obtained by experimental tests over a real traffic monitoring system. Although the model is proposed for a generic traffic monitoring system, in this section, the model is adapted and validated for the prototype called Ksensor. We expect to continue with the model validation over other platforms in the near future. It is also worth mentioning that some initial values are needed to assess

the theoretical model. They have been extracted from experimental measurements of Ksensor too.

4.1 Test Setup

Our hardware setup consists, as Fig. 8 shows, of four computers: one for traffic generation (injector), a second one for capturing and analyzing the traffic (Ksensor), a third one for packet reception (receiver) and the last one (manager) for managing, configuring and launching the tests. All they are physically connected to the same Gigabit Ethernet switch.

The basic idea is to overwhelm the system under test, Ksensor, with high traffic generated from the injector. In order to inject traffic bursts, we have installed an Endace 4.3GE DAG card [11] on the injector. Regarding software, we use a testing architecture designed by our research group [12]. The computer called manager is in charge of doing all the necessary tasks in order to automate the tests and measure the performance metrics of interest.

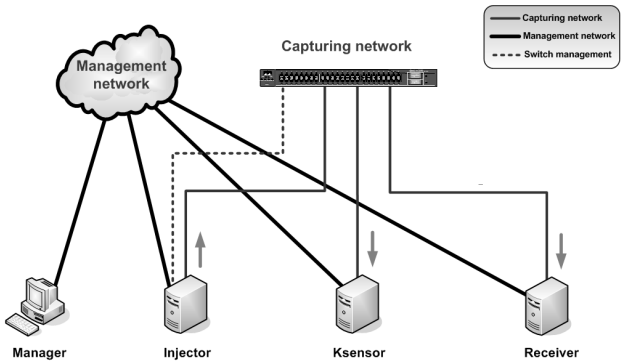


Figure 8. Hardware setup for validation tests.

The prototype Ksensor is a two-processor probe. One of the CPUs is responsible for capturing all the possible packets and analyzing some of them, whereas the other CPU is responsible only for analyzing packets.

4.2 Experimental Estimation for Certain Input Parameters of the Model

The model explained in Section 3 requires some input parameters such as μ_{C1}, μ_{C2}, μ_{C3}, μ_F, μ_A, p_{skb}, p_{enq} and q_a. We are referring to the service rates and probabilities that appear in the model (see Fig. 3) and are needed to obtain theoretical results.

At the end of every test, the manager (see Fig. 8) collects the measurements from Ksensor and the injector. Based on some of these experimental measurements (e.g. mean time consumed by a packet in every stage identified in the model, number of captured packets in softIRQ), we can estimate the model's input parameters. For example, the service rates μ_{C1}, μ_{C2}, μ_{C3}, μ_F and μ_A are computed as the inverses of the mean times experimentally obtained.

We have made tests varying the packet injection rate between 49487 and 1488000 packets per second, with packets of 40 bytes mean length and variable analysis loads (null, 1K, 5K, 25K) and we have observed that both the service rates and the probabilities are dependent on the injection rate and the analysis load. The exception is the probability q_a, because it is set before the test starts.

4.3 Performance Measurement Evaluation

After calculating the necessary values of the input parameters, the analytical expressions of Section 3 are assessed numerically. Table I and Fig. 9 illustrate this process showing, as an example, some numerical values for the case of 1K analysis load.

First, we have some experimental measurements obtained from Ksensor and the injector (e.g. the injection rate λ and the number of captured packets in softIRQ in Table I). Based on those measurements, we calculate the needed input parameters like the probability p_{enq} (see Table I).

TABLE I. EXAMPLES OF NUMERICAL VALUES USED IN THE EVALUATION OF THE ANALYTICAL MODEL

Experimental measurements from Ksensor and the injector			**Input param.**	**Intermediate result**	
Injection λ (pps)	***Captured Packets***	***Test seconds***	**p_{enq}**	**$\mu_{TMS}(1)$ (pps)**	**$\mu_{TMS}(2)$ (pps)**
49487	11924238	240,97	0,00	210690	383526
151298	36457717	240,97	0,00	210195	382273
255954	61678719	240,97	0,00	210546	383161
356470	85898013	240,98	0,00	213581	390861
453538	100799835	240,97	0,08	219462	399052
586642	100799835	240,97	0,29	216803	403453
664903	100037774	240,97	0,36	218532	405820
906022	105326168	240,97	0,51	224472	417279
1244992	104778112	240,97	0,65	223863	418296
1488000	108023963	245,25	0,70	224372	418332

Values for the case with packets of 40 bytes mean length and 1K analysis load in Ksensor (two-processor probe)

After that, the Norton equivalent of the traffic monitoring system is obtained. As explained before, it is necessary to compute the expressions of the repeated topology several times. Since we validate the model for Ksensor, the values of the parameters related to the number of processors are K_c=1 and K_f= K_a=2. Regarding the probability q_a, every test has been configured with q_a=1. Table I shows the values of the Norton equivalent of the traffic monitoring system as intermediate results. As they are state-dependent service rates, they have two values, for n=1 and n=2.

Finally, applying the solution for the closed network model with incoming traffic, we get the system throughput and it can be plotted like in Fig. 9.

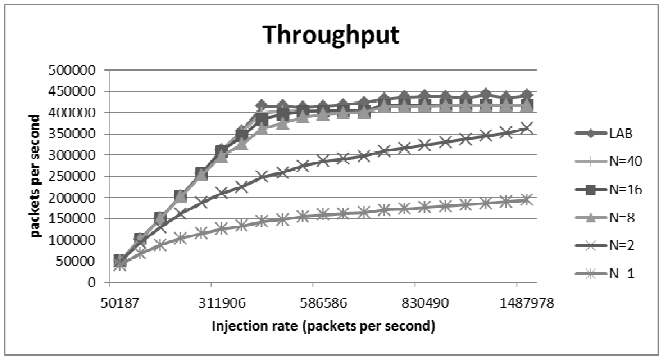


Figure 9. Theoretical and experimental throughputs with 1K analysis load in Ksensor (two-processor probe).

As mentioned, the model's main result is the theoretical throughput and, as seen in Fig. 9, it can be compared to the throughput measured by the experimental tests (LAB in Fig. 9). As we can observe, with N=40, the results are nearly the same. This process, which has been illustrated for the case of 1K analysis load, has also been done for the other cases (null, 5K and 25K). Acceptable results are obtained too.

5 Conclusions and Future Work

In this paper we have presented and analytical model that represents multiprocessor traffic monitoring systems. The model is generic and quantifies the system performance.

Initially, we detect experimentally an apparently strange behavior of the parameter called softIRQ times per packet in the traffic monitoring system. This addresses us to set out a model based in a closed queuing network which considers the capturing, filtering and analysis stages as well as the possible packet losses while the packet goes from the network to the monitoring application. We obtain the model's analytic solution, identifying a topology repeated at different levels of abstraction and applying the Norton equivalent to simplify the model. Then, the model is validated comparing theoretical results with experimental measurements over a prototype called Ksensor. In the validation process we make use of a testing architecture that not only measures the performance, it also provides values for some necessary input parameters of the mathematical model. As seen in the validation section, the obtained results are acceptable. Therefore, the model is able to represent precisely the behavior experimentally observed and also to evaluate the performance of the network traffic monitoring system, considering the most representative parameters like throughput, number of processors, analysis load and so on.

Moreover, the model is useful to interpret the variability of the processing time per packet in the capturing stage. The key is the parameter identified as p_{enq} in the model. In this work which combines the analytical study with experimental measurements, we have observed that this parameter is the most significant in the packet losses of the capturing stage. So, with high network rates, packet losses can be important and the smaller number of packets that reach the monitoring application require, in average, less CPU time.

Despite the fact that the conclusions have been satisfactory with regard to the behavior of the model, there are some aspects to be considered in the near future:

- We have presented a generic model for a traffic monitoring system, but we have only validated with the prototype Ksensor. We expect to adapt and validate the model to other platforms with more than two processors and over higher speed networks. We are especially interested in multicore systems and software probes under virtual machines.
- We have assumed Poisson processes and exponential service times and the results have been acceptable. However, we consider interesting to study the application of other distributions

All in all, we believe this work is one step for a better understanding of co-locating capturing process and monitoring applications. We can understand how to best process packets and whether new modeling techniques can be applied to perform network measurement.

6 References

[1] L. Deri, "Improving passive packet capture: beyond device polling" in Proceedings of SANE 2004, Amsterdam, The Netherlands, 2004.

[2] F. Fusco and L. Deri, "High speed network traffic analysis with commodity multi-core systems", in Proceedings of Internet Measurement Conference (IMC 2010), Melbourne, Australia, 2010.

[3] M. Smith and D. Loguinov, "High-performance internet-wide measurements on Windows", in Proceedings of Passive and Active Measurement Conference (PAM'10), Zuritz, Switzerland, 2010.

[4] A. Muñoz, A. Ferro, F. Liberal and J., Lopez, "A Kernel-Level Monitor over Multiprocessor Architectures for High-Performance Network Analysis with Commodity Hardware" in SensorComm 2007, Valencia, Spain, 2007.

[5] C. Benvenuti, "Understanding Linux Network Internals". O'Reilly Media, 2006.

[6] V. Bhaskar, "A closed queuing network model with multiple servers for multi-threaded architecture". Computer Comunications vol. 31 no. 14 pp. 3078-3089, 2008

[7] A. Roy, M.I. Islam and M.R. Amin, "MMPP+M/D/1 Traffic Model in Video-Data Integrated Service under ATM System" in IACSIT International Journal of Engineering and Technology, vol. 3, no. 6, pp. 615-620, 2011.

[8] E. Altman, K. Avratchenkov and C. Barakat, "A stochastic model for TCP/IP with stationary random losses", in ACM SIGCOMM 2000, Stockholm, Sweden, 2000.

[9] W. Leland, M. Taqqu, W. Willinger and D. Wilson, "On the self-similar nature of Ethernet traffic"in IEEE/ACM Transactions on Networking, vol. 2, no. 1, pp. 1-15, 1994.

[10] K.M. Chandy, U. Herzog and L.S. Woo, "Parametric Analysis of Queueing Networks Learning Techniques", IBM J. Research and Development, vol. 19, no. 1, pp. 43-49, 1975.

[11] Endace DAG Cards Enterprise Network Monitoring Tools, http://www.endace.com

[12] A. Pineda, L. Zabala and A. Ferro, "Network Architecture to Automatically Test Traffic Monitoring Systems". Mosharaka International Conference on Communications and Signal Processing (MIC-CSP2012), Barcelona, Spain, 2012.

A Power Controlled MAC Protocol with Improved Throughput for Ad hoc Networks

Santosh Kumar Yadav, A. K. Sarje
Department of Electronics and Computer Engineering,
Indian Institute of Technology Roorkee,
Roorkee, Uttarakhand, India
santosh.amh@gmail.com
sarjefec@iitr.ernet.in

Abstract - *Most of the devices in wireless ad hoc networks use battery power as the energy source; it makes the energy consumption as an important issue for such networks. Though energy maintenance can be performed at each layer but in the past, most of the researchers have focussed on MAC layer to minimize the energy consumption. The existing power controlled MAC protocols stress on reducing the energy consumption but they suffer from the disadvantage of throughput degradation of the network. In this paper, we have proposed a power controlled MAC protocol for wireless ad hoc networks that not only reduces the power consumption of each node in the network but also improves the throughput of the network. Our protocol involves two phases of work. Initially the protocol minimizes the power consumption of each node by sending the RTS packet with maximum power and CTS, DATA, ACK packets with minimum necessary power. DATA packets are periodically sent with maximum power to avoid unnecessary collisions. In the second phase, to improve the throughput of network, the spatial reuse is increased by changing the virtual carrier sensing mechanism of IEEE 802.11 protocol.*

Keywords: *ad hoc networks, energy consumption, MAC protocol*

1 Introduction

Ad hoc networks [1] are the wireless networks in which computing or electronic devices (referred as 'nodes') join together dynamically for transmitting the information from one node to another node in the form of data packets. The main characteristic which differentiates ad hoc networks from its counterpart infrastructure based network is the absence of any base-station in the ad hoc networks.

Wireless nodes use battery as the source of energy. As the battery power for any node is limited, it gives rise to the need of minimizing the battery power consumption for each node in the network. Minimizing the energy consumption of each node is the key to keep the ad hoc network functional for a long period of time.

Though energy consumption can be handled at each network layer, most of its maintenance is done at medium access control (MAC) layer. There are mainly two categories of MAC protocols for reducing the energy consumption at each node in ad hoc networks; first being 'Power saving MAC protocols' [2], [3] and second 'Power controlled MAC protocols' [4], [5], [6]. The first category i.e. 'Power saving MAC protocols' uses the concept of alternative sleep and wake up modes for wireless nodes. The nodes in ad hoc network remain in one of the three states: active, idle or sleep. Since power consumption in sleep state is less compared to other two states, we keep the nodes in sleep mode which are not participating in data transmission. On the other hand; in a network, power is consumed during computation and transmission of packets. But the computation power is negligible as compared to transmission power; hence efforts are made to reduce the transmission power. This is what 'Power control MAC protocols' uses and sends different packet types with different transmission power.

In this paper, we have focused on 'Power controlled MAC protocols' in which varying transmission power is used for transmitting the packets depending on whether the packet is RTS (request-to-send), CTS (clear-to-send), DATA or ACK (acknowledgement). In the past, most of the Power controlled MAC protocols focused on reducing the energy consumption of the wireless nodes without considering the aspect of network throughput which degraded on applying those power control techniques.

Our proposed power control MAC protocol not only reduces the power consumption of each wireless node of the network but also achieves improved throughput as compared to previous power controlled protocols. We have made some modifications in the virtual carrier sensing (VCS) scheme of IEEE 802.11 protocol to improve the spatial reuse of the network allowing more number of nodes to transmit frames at a time, which in turn, improves the throughput.

Our paper starts with a brief introduction of MAC protocols for ad hoc networks and the issues associated with them. Section 2 gives related works on power controlled MAC protocols. Section 3 covers the overview of IEEE 802.11

protocol which has been used as the base for power controlled MAC protocols. In section 4 our proposed power controlled MAC protocol is described. Simulation and results have been given in section 5. Finally, section 6 concludes the paper.

2 Related Works

Most of the existing Power control MAC protocols are based on IEEE 802.11 protocol. It uses CSMA/CA (carrier sense multiple access with collision avoidance) scheme, where carrier sensing is done using physical (air interface) as well as virtual carrier sensing. IEEE 802.11 involves four kind of packets namely RTS, CTS, DATA and ACK. Each packet's header contains a duration field that tells the time for which the transmission will take place. This duration field is used by virtual carrier sensing scheme to avoid collisions. Since IEEE 802.11 transmits each packet with the same maximum default power without taking any consideration of packet type, the energy consumption of each node is very high.

The PCMA protocol [8] allows different nodes in the network to send the packets with different transmission power levels. It uses four kinds of packets namely RPTS (Request-power-to-send), APTS (Acceptable-power-to-send), DATA and ACK. It follows the sequence RPTS-APTS-DATA-ACK for sending DATA packets. It uses two separate channels: *data channel* for RPTS-APTS-DATA-ACK sequence and *busy tone channel* for sending the busy tones while the node is receiving DATA packets. It uses busy tones to handle Hidden Node Problem instead of RTS-CTS handshake. Whenever a node is receiving DATA packets, busy tones are sent periodically by receiving node. Busy tone acts as a noise tolerance advertisement that can be tolerated by receiver. The drawback of PCMA is that it requires a separate signalling channel for busy tones. It also creates asymmetric links between nodes because each node sends packets with different power levels so it is possible that some node N1 can reach to any node N2 but not vice versa.

As a modification to IEEE 802.11 protocol, other power controlled protocols use maximum default power for transmitting the RTS and CTS packets while DATA and ACK packets are sent by minimum required power. This scheme has been termed as 'Basic Power Control Protocol' [5], [6]. Hence, it is possible that nodes in the CSZ (carrier sensing zone) that sense RTS-CTS transmission may not be able to sense any signal during DATA-ACK. Thus, when these nodes start a new transmission by sending RTS with max power, collision may occur with both the DATA packet at receiver as well as with the ACK packet at sender. This triggers retransmission of packets which results in more energy consumption and reduced throughput.

Power Control MAC (PCM) protocol [5] was given to remove the deficiency of Basic power control protocols. It is similar to the Basic power control protocol, but in addition to improving the throughput of the network, the DATA packets are sent with the maximum default power periodically.

F-PCM protocol [6] uses the fragmentation technique along with PCM protocol to further improve the throughput. It allows fragmenting of large DATA and ACK packets. Here also, the RTS and CTS packets are sent with maximum default power while the DATA and ACK packets are sent with minimum necessary power. But during the beginning of sending the DATA packet, the initial DATA fragments are sent with maximum power. A similar approach is applied to ACK packets.

Thus, by analysing the previous power control MAC protocols, we notice that though they reduce the energy consumption of each node but none of them improves the throughput of network in comparison to IEEE 802.11 MAC protocol. Our work includes both, reducing the power consumption of each node and improving the throughput of the networks.

3 IEEE 802.11 MAC Protocol

To understand IEEE 802.11 protocol [5], [6], some of the basic terms have been defined as follows:

(i) *Transmission range*: The range in which any other node can both receive the packets as well as decode the information contained in the header of packets correctly, those were sent by the sender. For example, duration field contained in the header of a packet.

(ii) *Carrier sensing range*: The range in which any other node can sense the packets sent by the sender. The transmission range is a subset of carrier sensing range.

(iii) *Carrier sensing zone*: The zone which is outside the transmission range but within the carrier sensing range of the sender is called carrier sensing zone. In this zone, any node can sense the transmission of packets by sender but cannot decode the packet's header correctly.

IEEE 802.11 protocol is based on CSMA/CA scheme and it uses four kinds of packets RTS-CTS-DATA-ACK. The virtual carrier sensing scheme uses duration field of these packets to determine the time for which the current transmission would remain continues and the channel would remain busy. To avoid any collisions, each node maintains an NAV (Network Allocation Vector) in which the remaining time of the current transmission is maintained. The NAV handling at each node is done as shown in Fig. 1.

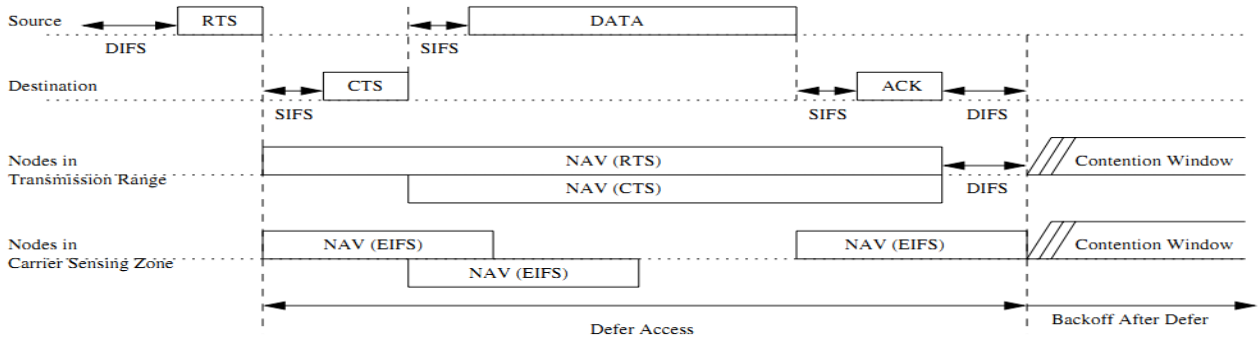


Fig. 1. RTS-CTS-DATA-ACK handshaking process in IEEE 802.11 [5]

4 Proposed Power Controlled MAC Protocol with Improved Throughput

The existing power control MAC protocols reduce the energy consumption of each node but they fail to obtain the high throughput compared to IEEE 802.11 protocol. The purpose of our work is to design a new power control MAC protocol that minimizes the energy consumption of each node as well as improves the throughput of the network compared to IEEE 802.11 protocol. The work of the paper is divided into two phases:

Phase I: Design a power controlled protocol that minimizes the energy consumption of each node.

Phase II: Associate the designed power control MAC protocol with the modified VCS scheme of IEEE 802.11 protocol for increasing the spatial reuse that will improve the throughput of the network.

4.1 Minimizing the Energy Consumption of Each Node

Like most of the power control MAC protocols, in our paper also, the reduction of power consumption for each node can be achieved by sending the packets with optimum power (the minimum energy to reach the packets at their receiver). IEEE 802.11 sends each kind of packet irrespective of RTS, CTS, DATA or ACK with the same maximum power. Other power controlled protocols send the RTS and CTS packets at maximum power level while DATA and ACK packets are sent with minimum necessary power that results in the degradation of throughput due to increased number of collisions. Hence, in our protocol, considering the throughput improvement and minimizing the energy consumption, we send the RTS packet with the maximum power but all other packets i.e. CTS, DATA and ACK are sent with optimum power. DATA packets are periodically sent with maximum power to avoid unnecessary collisions.

Implementation of our protocol requires the addition of a new field *P_RTS* to the RTS packet and *P_DATA* field to the CTS packet; where *P_RTS* is the power level at which RTS is sent and *P_DATA* is the optimum power level at which the DATA packets are required to be sent for successful receiving. CTS and ACK are also sent with the same optimum power. The structures of RTS and CTS packets are as shown in the Fig. 2 for IEEE 802.11 and our proposed protocol:

Frame control	Duration	RA	TA	CRC

RTS frame format in IEEE 802.11 protocol

Frame control	Duration	RA	TA	CRC	P_RTS

RTS frame format in proposed protocol

Frame control	Duration	RA	CRC

CTS frame format in IEEE 802.11 protocol

Frame control	Duration	RA	CRC	P_DATA

CTS frame format in proposed protocol

Fig. 2. Structure of RTS and CTS frames

The working procedure of our protocol for minimizing the energy consumption is as follows:

- The transmitter node broadcasts the RTS packet at a power level *P_RTS*. Here, *P_RTS* is equal to the maximum transmitted power used in IEEE 802.11 protocol.
- The receiver receives the RTS packet at a received power *Pr*. For two ray propagation model, *Pr* at a distance *d* is calculated as follows

$$Pr = \frac{Pt \times Gt \times Gr \times Ht^2 \times Hr^2}{d^4 L}$$

  Where, *Pt* is the transmitted power, *Gt* and *Gr* are the transmitter and receiver gains, *Ht* and *Hr* are the heights of transmitter and receiver antennas and *L* is the loss factor.
- The receiver extracts *P_RTS* from the received RTS packet and calculates the value of *P_DATA* as follows and appends it in the CTS packet

$$P\_DATA = \frac{P\_RTS}{\mathrm{Pr}} \times RxThreshold$$

  Where *RxThreshold* is the receiving threshold of signal strength at which the receiver can decode the signal.
- The receiver sends the CTS packet at a power level *P_DATA*.

- The transmitter sends the DATA packets at the power level *P_DATA* extracted from the received CTS packet.
- The receiver after receiving the DATA packet sends the ACK packet at same power which was used to send the CTS packet.

DATA packets are periodically sent with maximum power to avoid the unnecessary collisions with ACK packets at source node.

4.2 Throughput Improvement over IEEE 802.11

As mentioned in the above section, the nodes use different power level for different transmissions based on the packet type which may degrade the throughput of the network. The throughput of a network can be improved by increasing the spatial reuse in the network. In order to increase spatial reuse, we modify the VCS (virtual carrier sensing) scheme of IEEE 802.11 protocol.

In 802.11 VCS scheme, if a node overhears an RTS or CTS packet, it assumes the channel as busy and sets its NAV. Thus, if the node which overhears has any packet to send, it defers the transmission for the time of duration field's value extracted from the overheard packet. In our protocol, we improve this VCS scheme.

A node can overhear an RTS packet only or a CTS packet only or both RTS and CTS packets. In our protocol, since CTS transmission range is less compared to RTS transmission range, there is a chance that a node overhears the RTS packet only and it has to send a CTS packet, then it sends the CTS packet immediately. It doesn't affect the ongoing transmission. For example, consider Fig. 3; suppose node N1 has data to send to node N2. So node N1 sends an RTS packet to node N2. Since N3 is in transmission range of node N1, node N3 will overhear the RTS packet. After receiving RTS from node N1, node N2 will respond by a CTS packet. Since the transmission range of its CTS is small, it will not be overheard by node N3 and therefore N3 will not set its NAV. Now, if node N3 receives an RTS packet from node N4, it can respond immediately with a CTS packet. Thus, it will improve spatial reuse because more nodes can send packets at a time which, in turn, will improve the throughput of the network.

To achieve our goal, we propose some modifications in the VCS scheme of IEEE 802.11 which are as follows

- As IEEE 802.11 VCS scheme uses NAV, we introduce one more parameter NAVR along with NAV.
- If a node in the transmission range overhears an RTS packet, it sets its NAV to a slot time and NAVR to the value of duration field extracted from the RTS packet. The slot time includes the time to sense the channel creating and transmitting the response.

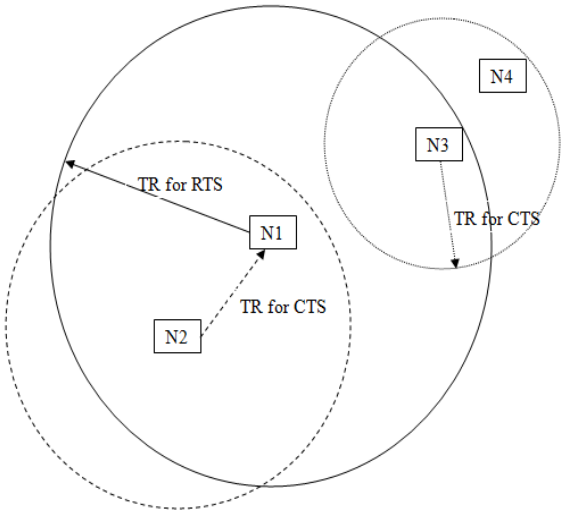


Fig. 3. An example to show transmission ranges of RTS and CTS

- If a node overhears a CTS packet before the NAV expires, it sets the NAV to the value of duration field extracted from the CTS packet.
- Suppose, a node in the transmission range wants to send a packet then first of all, it checks the packet type. If it is an RTS packet, it can send only if NAVR has expired, otherwise, it has to wait for backoff time. While, if it is a CTS or DATA packet, it can send only if NAV has expired. Like IEEE 802.11, ACK packets are sent immediately in our protocol also.

5 Simulation and Results

We have implemented our proposed protocol on NS-2.34 network simulator [7]. The metrics, on which we have focused, are energy consumption of each node and throughput of the network. We setup the scenario of 50 nodes placed randomly in a terrain size of 800m X 800m. The routing protocol used is DSDV. The packet size used is 512 bytes. We have simulated the same scenario for both IEEE 802.11 and our proposed protocol and observed the changes in energy consumption and throughput. The simulation parameters is be tabulated as below

TABLE I
Simulation Parameters

Terrain size	800m X 800m
Number of nodes	50
Application	CBR
Packet size	512 bytes
Routing protocol	DSDV
Antenna model	Omni directional
Propagation model	Two ray ground

Fig. 4, Fig. 5 and Fig. 6 show the results obtained at the end of simulation. Fig. 4 shows that the energy consumption of each node in our proposed protocol is less compared to IEEE 802.11. To compare the throughput of the network in IEEE 802.11 and our proposed protocol, we run the same scenario for different packet sizes. Fig. 5 shows the results of total energy consumption in the network system at the end of simulation on varying the packet sizes. It depicts that for each kind of packet size, the total energy consumption in the network for our proposed protocol remains lesser than IEEE 802.11 protocol. Fig. 6 clearly shows an improvement in the throughput of the network for our proposed protocol in comparison to IEEE 802.11.

6 Conclusions

In this paper, we have proposed a power controlled MAC protocol with improved throughput for wireless ad hoc networks. MAC protocols given in the past were mainly focused on reducing the energy consumption only. But the disadvantage associated with those protocols is the degradation in the throughput of network while reducing the energy consumption of each node. We have developed a protocol which improves the throughput of the network along with minimizing power consumption of each node. For this purpose, we first changed the power levels of transmissions for different kind of packets and associated it with the modified VCS scheme of IEEE 802.11 protocol to increase the spatial reuse allowing more number of simultaneous transmissions which ultimately results in the increase in throughput of the network. The protocol has been simulated on NS-2.34 and the results obtained at the end of simulation satisfy the goal of our paper.

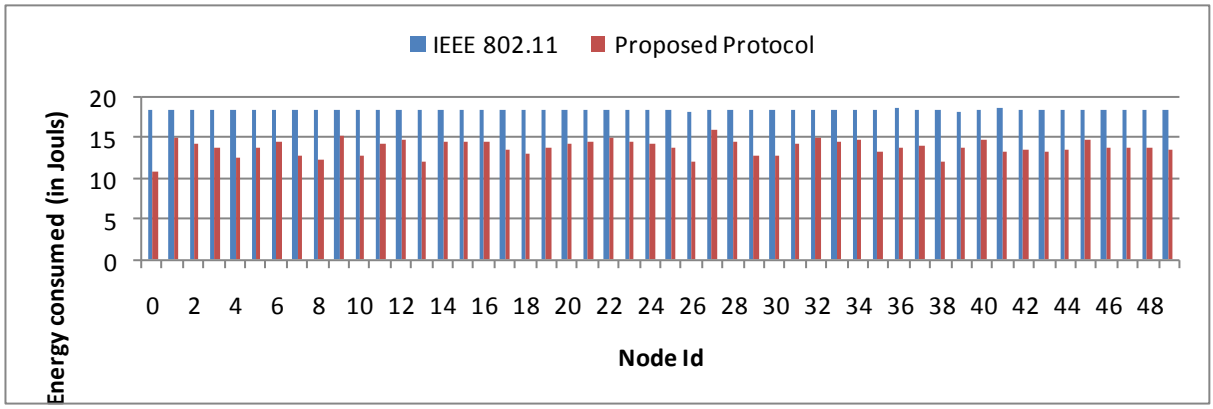


Fig. 4. Energy consumption of each node in IEEE 802.11 Vs our proposed protocol

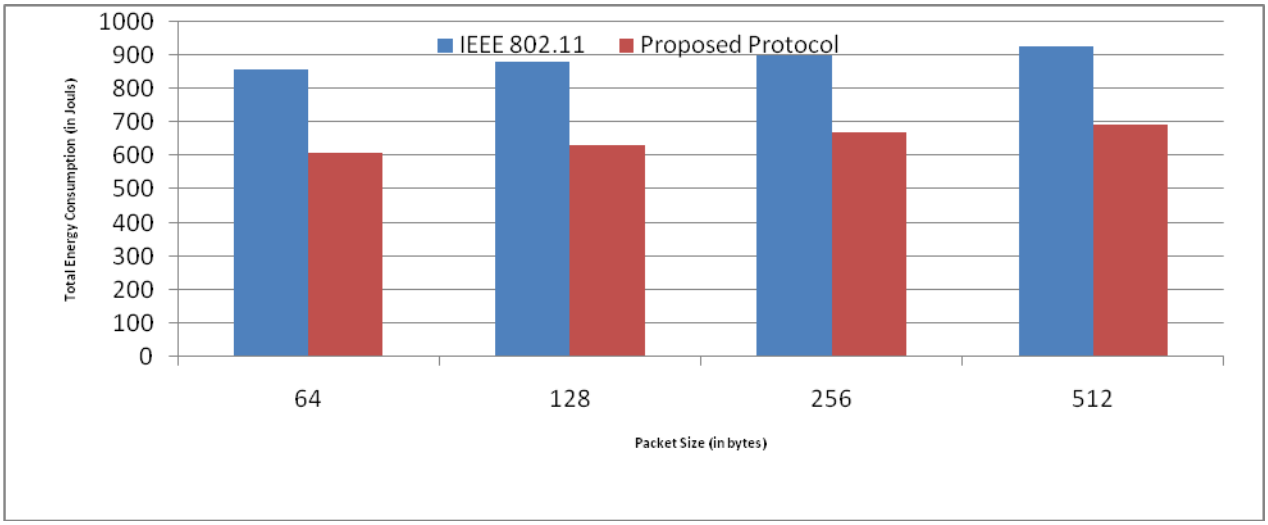


Fig. 5. Total energy consumption in IEEE 802.11 Vs our proposed protocol

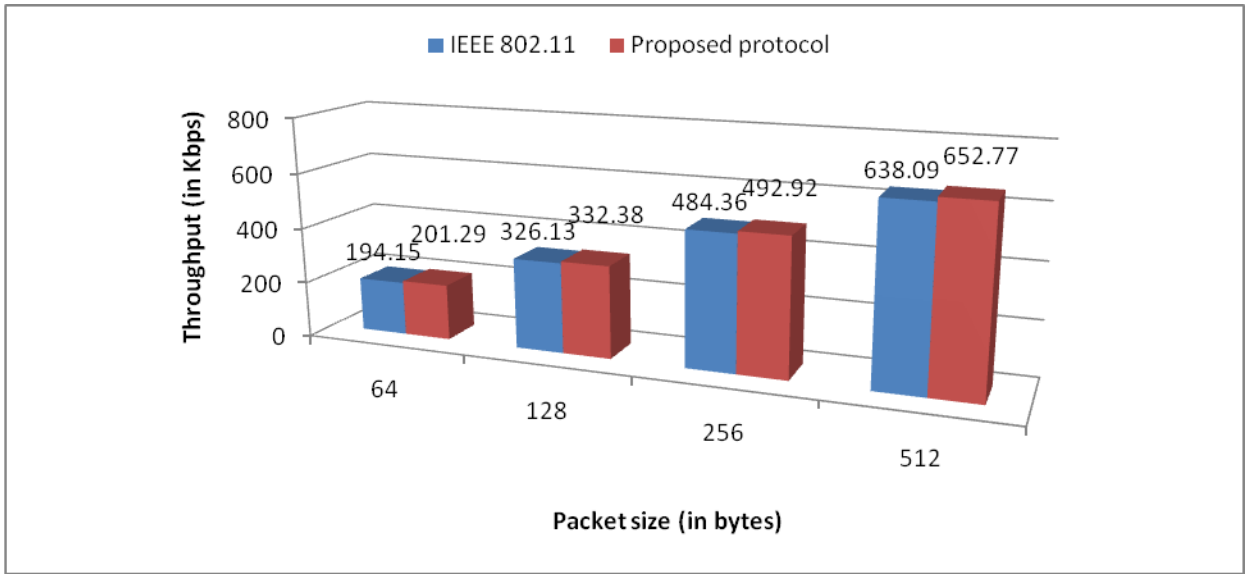


Fig. 6. Throughput improvement for different packet sizes in our proposed protocol w.r.t. IEEE 802.11

7 References

[1] J.H. Schiller. "Mobile Communications" ; 2nd ed., Addison-Wesley, Pearson Education Limited Publishers, pp. 330-345, 2003.

[2] S. Singh and C.S. Raghavendra. "PAMAS – Power aware multi-access protocol with signaling for ad hoc networks" ; ACM Comput. Commun, vol. 28, no. 3, pp. 5-26, 1998.

[3] H. Woesner, J.P. Ebert, M. Schlager and A.Wolisz. "Power-saving mechanisms in emerging standards for wireless LANs: The MAC level perspective" ; IEEE Personal Communication, vol. 5, no. 3, pp. 40-48, 1998.

[4] J.P. Monks, V. Bharghavan and W. Hwu. "A power controlled multiple access protocol for wireless packet networks" ; Proceedings of the IEEE INFOCOM, pp. 219-228, April 2001.

[5] E.S. Jung and N.H. Vaidya. "A power control MAC protocol for ad hoc networks" ; ACM International Conference Mobile Computing and Networking (MOBICOM), pages 36-47, September 2002.

[6] D. Kim and C.K. Toh. "F-PCM: A fragmentation-based power control MAC protocol for IEEE 802.11 mobile ad hoc networks" ; Wireless Communications and Mobile Computing, vol. 6, no. 5, pp. 727-739, August 2006.

[7] (2011) The ns-2 website. [Online]. Available: http://nsnam.isi.edu/nsnam/index.php/User_Information/

Secure Data Collection for Wireless Sensor Networks

Haengrae Cho[1] **and Soo-Young Suck**[2]
[1]Department of Computer Engineering, Yeungnam University, Republic of Korea
[2]Department of R&D, Gyeongbuk Institute of IT Convergence Industry Technology, Republic of Korea

Abstract— *Wireless sensor networks (WSNs) are used to collect various data in environment monitoring applications. The most comprehensive way of data collection is to make every sensor node report periodically its sensing data to a base node. To reduce the energy consumption due to excessive communication, the network is partitioned into a set of spatial clusters with similar sensing data. For each cluster, only a few sensor nodes (samplers) report their sensing data to the base node. The base node may predict the missed data of non-samplers using the spatial correlation between sensor nodes. The spatial clustering is vulnerable to internal security threat such as node compromise. If the samplers are compromised and report incorrect data intentionally, then the WSN should be contaminated rapidly due to the process of data prediction at the base node. In this paper, we propose intrusion detection algorithms for secure data collection in the WSN. The algorithms consist of cluster head monitoring and member node monitoring to cope with security attacks where either a cluster head or member nodes are compromised.*

Keywords: Wireless Sensor Network, Data Collection, Spatial Clustering, Security, Attack Detection

1. Introduction

A wireless sensor networks (WSN) typically consist of a large number of small battery-powered sensor nodes. In order to sustain sensor nodes to run for a long period, it is critical to save energy in sensor operations [1]. The primary functions of WSN are to collect data for observation and analysis of physical phenomena. There are two types of data collection in WSN: *event-based* and *periodic* approach [5]. In event-based data collection, sensor nodes are responsible for detecting and reporting (to a base node) events such as spotting moving targets. They perform local filtering and sometimes collaborate with each other to detect events. On the other hand, in periodic data collection, every node reports periodically its sensing data to the base node. Many researches prefer the periodic approach because it enables arbitrary data analysis at the base node [3], [5].

Extracting the vast amounts of data generated by large-scale, high-density WSN can cause a wide range of problems. Sensing always and transmitting every data would cause sensor nodes to drain their batteries soon. Furthermore, the limited communication bandwidth prevents all the acquired data from being propagated successfully toward the base node. This means that we need an energy-efficient way of data collection to prolong the lifetime of WSN by keeping the energy consumption at minimum.

Spatial clustering is a representative way of saving energy in periodic data collection [6], [9]. It partitions the network into a set of clusters where a cluster includes sensor nodes with similar sensing data. For each cluster, only a few sensor nodes (samplers) report their data to the base node. All the rest of sensor nodes can save their energy by keeping in sleep mode. The base node may predict the missed data using the spatial correlation between sensor nodes. To balance the energy consumption, sensor nodes within a cluster can share the workload equally.

The WSN is vulnerable to security threats both external and internal due to unreliable wireless channels, unattended operation of sensor nodes, and resource constraint [2], [12]. *Node compromise* is a major type of internal attacks. Compromised sensor nodes release all the security information to the adversary. Then, the adversary can easily launch internal attacks with data alteration, message negligence, selective forwarding, and jamming [4], [8], [10]. Note that the node compromise is especially problematic for periodic data collection applications, where only the samplers may report data to the base node. If the samplers are compromised and report incorrect data intentionally, then the WSN should be contaminated rapidly due to the process of missing data prediction at the base node. This means that detecting and defending against node compromise are inevitable tasks to guarantee the correctness of data collection at WSN.

In this paper, we propose intrusion detection algorithms for secure data collection in the WSN. The algorithms consist of *cluster head monitoring* and *sampler monitoring* to cope with internal security attacks where either a cluster head or member nodes are compromised. In the cluster head monitoring, neighbor nodes of a cluster head collaboratively monitor their cluster head. The member node monitoring algorithm is divided into two variants: *monitoring by neighbors* (MBN) and *monitoring by cluster head* (MBCH). They are based on spatial clustering and try to detect compromised nodes with energy-efficient manner. The MBN explores the spatial correlation with a sampler and its neighbors. Neighbors have a role to watchdog. They listen promiscuously to the sampler's broadcasting transmission and monitor whether the sampler is compromised or not. The MBCH does not impose any additional roles to sensor nodes for security. Instead, the cluster head has to monitor the transmitted data

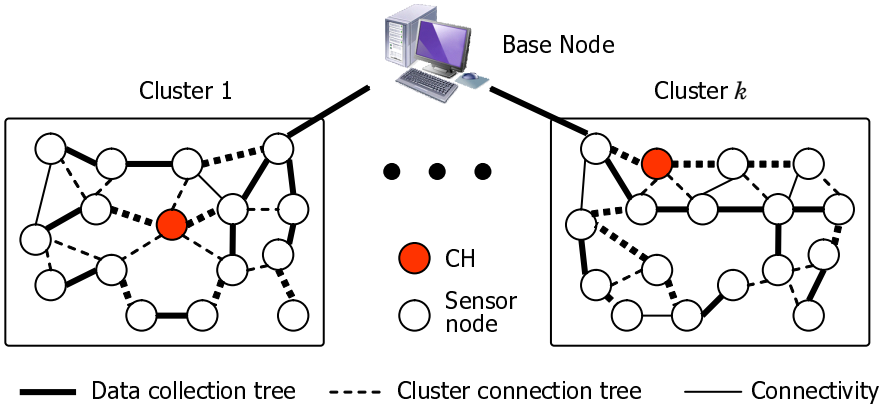


Fig. 1: Model of a WSN

of samplers and compares them with the sensor readings of non-sampler nodes.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 describes our model of the WSN and the data collection. Section 4 describes the proposed algorithms in detail and discusses some performance issues. Finally, Section 5 concludes this paper.

2. Related Work

Most of previous intrusion detection algorithms proposed for the WSN did not consider the underlying data collection architectures [4], [8], [10]. The two exceptions are [7] and [11]. Authors of [7] partition the WSN into several δ-groups by extending the distributed spatial clustering algorithm [9]. Sensor nodes within the same group are physically close to each other and their sensed data are dissimilar by at most δ. To detect compromised nodes, they partition the δ-group into equal-sized sub-groups. Each sub-group monitors the entire δ-group in turn to reduce the total power consumption. However, they did not consider the location information of monitoring nodes and reporting nodes. If a monitoring node is not located in the routing path between a reporting node and the base node, it cannot detect whether the reporting node is compromised or not. Furthermore, there is no central decision point and thus every node in the cluster has to decide the node compromise if alert messages are broadcast from the monitoring nodes.

Authors of [11] proposed a collaboration-based intrusion detection algorithm to detect and revoke compromised nodes in a cluster through less energy consumption. Similar to this paper, they propose separate algorithms for cluster head monitoring and member node monitoring. To monitor cluster head, all member nodes of a cluster are divided into several groups. Then every monitor group takes turns to monitor the cluster head in a cluster round time. The cluster head monitors its member nodes and is responsible to detect the compromised member nodes. Note that this is contrary to [7] where every node has to decide the node compromise. The main problem of [11] is that they did not present how to detect the misbehavior of each sensor node. This should be performed on the basis of several monitoring attributes about sensed data and communication behaviors [7], [8], [10]. Furthermore, for cluster head monitoring, they did not consider the location information of monitoring nodes and the cluster head. This should cause to decrease the detection accuracy.

Unlike previous cluster-based intrusion detection algorithms, our algorithm is completely integrated into the underlying data collection architecture. Our algorithm considers the WSN structure, cluster formation, and data collection procedures. This enables our algorithms to optimize energy consumption for message transmission and sampling.

3. Model of WSN and Data Collection

Figure 1 shows our model of a WSN. The WSN follows layered network architectures [5]. Let $S = \{s_1, \ldots, s_n\}$ be a set of all sensor nodes in the WSN. A *data collection tree* is used to propagate the sensed data of each node to the base node. The base node is the root of the data collection tree. The WSN is partitioned into disjoint clusters, where a sensor node is selected as a *cluster head* (CH). A *cluster connection tree* is used to establish the communication between the CH and the other nodes in the cluster. Each sensor node is assumed to be able to communicate only with its neighbors. The set of neighbor nodes of sensor node s_i is denoted by $nbr(s_i)$. The nodes that can communicate with each other form a *connectivity graph*. s_i is forced to sample periodically, and let $D[s_i]$ be a vector of sampled readings of s_i and L be its length. The correlation of two sensor nodes s_i and s_j are defined as [5]:

$$\text{Corr}(s_i, s_j) = \frac{(D[s_i] - \text{E}[D[s_i]]) * (D[s_j] - \text{E}[D[s_j]])^T}{L * \sqrt{\text{Var}(D[s_i])} * \sqrt{\text{Var}(D[s_j])}} \quad (1)$$

Definition 1. Two sensor nodes, s_i and s_j, are *strongly correlated* if $1 - |\text{Corr}(s_i, s_j)| \leq \delta$, where $0 \leq \delta \leq 1$.

Definition 2. A set of sensor nodes C is called a *cluster* it the following two conditions hold for every pair of $s_i \in C$ and $s_j \in C$: (1) s_i can communicate with s_j directly or via any nodes in C, and (2) s_i and s_j are strongly correlated.

The construction of clusters based on spatial correlation is an interesting research issue and has been studied by many researchers [3], [5], [6], [9]. In this paper, we assume that the WSN is already partitioned into several clusters as Figure 1 shows.

For each cluster, the CH maintains the correlation information for every pair of sensor nodes in the cluster. When a new cluster is established, the CH sends a data mean vector (E[D[s_i]]) and a data covariance matrix (Corr(s_i, s_j)$*$ $\sqrt{\text{Var}(D[s_i])} * \sqrt{\text{Var}(D[s_j])}$) for every pair of sensor nodes s_i and s_j to the base node. Each sensor node in the cluster periodically samples its sensor data and sends it to the CH. We call the period of sampling as a *forced-sampling period* (τ_f). The CH evaluates the degree of correlation periodically and starts a new cluster construction phase if sensor nodes in the cluster are not strongly correlated anymore.

Each node in a cluster becomes a sampler with a probability λ. Combining this randomized scheduling with the round robin scheduling, we can guarantee that at least one node becomes a sampler for each cluster. A sampler sends its sensing data to the base node for every τ_d through the data collection tree. Then the base node can predict the sensing data of non-sampler nodes with the data mean vector and the data covariance matrix [5]. We call τ_d as a *data-sampling period*. Note that we can save energy significantly by setting τ_f to be much longer than τ_d.

4. Secure Data Collection Algorithms

In this section, we first present a cluster head monitoring algorithm. Then we describe two sampler monitoring algorithms, *monitoring by neighbors* (MBN) and *monitoring by cluster head* (MBCH). Finally, we analyze our algorithms qualitatively.

4.1 Cluster Head Monitoring

In periodic data collection framework, the CH has a primary role and thus its security demand is higher than that of the other sensor nodes. The role of the CH can be summarized as follows.

- The CH constructs a cluster that consists of sensor nodes which are strongly correlated to that of the CH.
- After constructing the cluster, the CH sends a data mean vector (E[D[s_i]]) and a data covariance matrix (Corr(s_i, s_j) $* \sqrt{\text{Var}(D[s_i])} * \sqrt{\text{Var}(D[s_j])}$) for every pair of sensor nodes s_i and s_j to the base node. The base node will use them to derive the parameters of the probabilistic models used in predicting the values of non-sampler nodes.
- For each forced-sampling period, the CH evaluates the degree of correlation periodically and starts a new cluster construction phase if sensor nodes in the cluster are not strongly correlated anymore.
- The CH is responsible to detect compromised nodes in its cluster.

The first three roles of the CH correspond to the cluster maintenance. Detecting the misbehavior of the CH at the cluster maintenance is a challenge, since most decisions of the CH comes from the raw information of member nodes such as the history of sensor readings. This cannot be done simply by neighbor based detection of communication behavior [8], [10]. A potential solution is to assign several CHs in a cluster and to make them monitor with each other. However, member nodes must suffer from heavy communication overhead to report their sensor readings to every CH. We let this issue as a future work of this paper.

In this paper, we concentrate the last role of the CH. Some sensor nodes will report the monitoring information to the CH using the member node monitoring algorithm of Section 4.2. Then the CH checks the information and announces to every member node in the cluster when some sampler is compromised. This decision could be incorrect if the CH itself is compromised. Suppose that *nbr*(CH) is the set of neighbor nodes of the CH. Then each sensor node s_i in *nbr*(CH) executes the following steps.

1) For each data-sampling period, s_i overhears the monitoring information sent to the CH. It delivers the information to every other node in *nbr*(CH).
2) s_i also overhears the decision message sent from the CH. If the decision is different from that of itself, it decides that the CH is compromised and announces its decision to every other node in *nbr*(CH).
3) If more than a predefined percentage of sensor nodes in *nbr*(CH) decide that the CH is compromised, one of them reports it to the base node. Then the base node segregates the CH from the WSN by broadcasting the decision to every node and assigns a new CH for the cluster.

Note that neighbor nodes of the CH would drain its battery rapidly due to the monitoring task. The primary source of energy consumption is to make consensus among neighbor nodes. We can reduce the energy consumption by considering the type of false alarm. For examples, if we allow false negative errors where a correct node is determined as compromised, neighbor nodes need not exchange their decisions when the CH announces that some node is compromised. Furthermore, most data collection algorithms select new CH periodically to prolong the lifetime of the WSN. This means that the energy consumption of neighbor nodes can also be distributed to other nodes.

4.2 Sampler Monitoring

The CH has a role to decide whether a sampler is compromised or not. For each data-sampling period, samplers report their sensor readings to the base node through the data collection tree. If the CH does not locate on the routing path from the sampler to the base node, it cannot detect the compromised sampler. Hence, we modify the data collection process by reporting the monitoring information to the CH. In this paper, we propose two sampler monitoring algorithms. The algorithms have different strategies to select monitoring nodes that report to the CH and to define monitoring information to be reported.

4.2.1 Monitoring by Neighbors (MBN)

In the MBN, every neighbor node of a sampler has a role to a watchdog that monitors the message sent from the sampler. The following modifications of data collection procedures are required for the MBN to determine whether a sampler is compromised or not.

- The CH has complete location information for every sensor node in its cluster to identify neighbor nodes of the sampler.
- Each sensor node has a correlation vector to every other sensor node in the cluster.

Suppose that a sensor node s_s is selected as a sampler. s_s notifies itself to the CH[1], and the CH wakes up the sensor nodes in $nbr(s_s)$. For each node $s_i \in nbr(s_s)$, it reads its sensing data for every τ_d. Then s_i overhears the message from s_s no matter whether or not s_i is involved in the communication. If the sensor reading sent from s_s is not strongly correlated to that of s_i, s_i reports to the CH. If more than half sensor nodes of $nbr(s_s)$ report to the CH, the CH decides that s_s is compromised and reports it to the base node. After that, the base node ignores the data sent from s_s. Each sensor node also excludes the compromised node in selecting the next-hop forwarder to realize the secure routing.

4.2.2 Monitoring by Cluster Head (MBCH)

In the MBCH, the CH has a role to detect the compromised node. The following modifications of data collection procedures are required for the MBCH to determine whether a sampler is compromised or not.

- For each cluster, at least two samplers should be selected for every data-sampling period.
- Samplers are required to send its sensor readings to the CH.

In the MBCH, the CH performs the detection of compromised node by two ways: (1) comparison between samplers and (2) comparison between sampler and non-samplers. Suppose that two sensor nodes, s_i and s_j, are selected as samplers of a cluster. For every data-sampling period, s_i and s_j send their sensor readings to the CH. The CH then forwards the message to the base node only if they are strongly correlated. Otherwise, one of samplers could be compromised. In this case, the CH does not forward the message and waits until the next forced-sampling period to collect every sampling data and to determine the compromised node.

Note that two samplers are not enough if both of them could be compromised. To check if such condition happens, the CH compares the validity of samplers with sensor readings of every non-sampler node for some forced-sampling period. If the samplers are compromised, the CH reports them to the base node. Then the base node invalidates previous sensing data sent from the compromised samplers. If we increase the number of samplers, the WSN should be more secure at the cost of increased energy consumption. This shows an interesting tradeoff between energy consumption and security enforcement.

4.3 Qualitative Analysis

The performance of MBN and MBCH depends on several factors of WSN, such as network density, capacity of sensor nodes, sampling cost, and so on. In this section, we analyze the pros and cons of two algorithms for each factor.

- ***Network density***: The MBN is effective only if there are sufficient neighbor nodes for each sampler. This is because the majority vote is performed to determine if the sampler is compromised. If the number of neighbor nodes is not enough, the MBN may be exposed to the unsafe case when both the sampler and its neighbor nodes are compromised at the same time. On the other hand, the MBCH is less dependent on the network density due to the additional step of comparison at the forced-sampling period. Note that the MBN does not support the comparison at the forced-sampling period, since the sampler is not required to send its sensor readings to the CH.
- ***Capacity of sensor nodes***: The MBN requires that every sensor node can store correlation information to every other sensor node in the cluster. Furthermore, neighbor nodes of a sampler has to (1) sample at each data-sampling period, (2) overhear the message sent from the sampler, (3) calculates the correlation between itself and the sampler, and (4) report to the CH in case of correlation mismatch. This means that the MBN spends a lot of memory resources and computing resources of sensor nodes. On the other hand, the MBCH does not spend any additional memory space of sensor nodes for security enforcement. It just requires more samplers. However, since the CH detects the node compromise for itself, the CH can drain its energy more rapidly. This

[1]To force this procedure to compromised node, the CH may assign some unique id to the sampler in response to the notification. The base node should reject a message from the sampler if it does not contain the id.

means that the MBCH depends on the CH replacement algorithm to prolong the lifetime of WSN.

- ***Sampling cost***: If the sampling cost is not expensive, the overhead of the MBN to make neighbor nodes sample at every data-sampling period is not significant. In this case, maintaining additional samplers at the MBCH may take much overhead due to communication cost of the samplers. Note that the message overhearing at promiscuous mode of the MBN spends much less energy compared with the message transmission at samplers. However, if the sampling cost is high, sensor nodes of the MBN would spend more energy especially for the dense network.
- ***Detection time of compromised node***: The MBN can detect the comprised node as soon as the majority of neighbor nodes vote. On the other hand, the MBCH cannot determine the compromised node promptly until the next forced-sampling period. Note that the MBCH can reduce the delay if it operates many samplers and applies majority vote between them. However, many samplers should result in increased energy consumption and network traffic.

5. Conclusions

In this paper, we propose intrusion detection algorithms for secure data collection in the WSN. The proposed algorithms are composed of cluster head monitoring algorithm and sampler monitoring algorithm. The sampler monitoring algorithm is also composed of two sub-algorithms, monitoring by neighbors (MBN) and monitoring by cluster head (MBCH). They are based on spatial clustering and try to detect compromised nodes with energy-efficient manner. Unlike previous security algorithms for WSN, our algorithms consider the underlying data collection architectures. The security task is completely integrated to data collection algorithm. This enables our algorithms to optimize energy consumption for message transmission and sampling.

We are investigating quantitative analysis of the secure data collection using our simulation model. The simulation model is developed with CSIM package and implements variety of WSN configurations and workloads.

References

[1] G. Anastasi, M. Conti, M. D. Francesco, and A. Passarella, "Energy Conservation in Wireless Sensor Networks: A Survey," *Ad Hoc Networks*, vol. 7, pp. 537–568, 2009.

[2] X. Chen, K. Makki, K. Yen, and N. Pissinou, "Sensor Network Security," *IEEE Communications Surveys & Tutorials*, vol. 11, pp. 52–73, 2009.

[3] H. Cho, "Distributed Multidimensional Clustering based on Spatial Correlation in Wireless Sensor Networks," *Computer Systems Science and Engineering*, vol. 26, pp. 275–283, 2011.

[4] X. Du, "Detection of Compromised Sensor Nodes in Heterogeneous Sensor Networks," in *Proc. 2008 International Conference on Communications (ICC 2008)*, pp. 1446–1450.

[5] B. Gedik, L. Liu, and P. S. Yu, "ASAP: An Adaptive Sampling Approach to Data Collection in Sensor Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, pp. 1766–1783, 2007.

[6] T. D. Le, N. D. Pham, and H. Choo, "Towards a Distributed Clustering Scheme based on Spatial Correlation in WSNs," in *Proc. 2008 International Wireless Communications and Mobile Computing Conference (IWCMC 2008)*.

[7] G. Li, J. He, and Y. Fu, "Group-based Intrusion Detection System in Wireless Sensor Networks," *Computer Communications*, vol. 31, pp. 4324–4332, 2008.

[8] F. Liu, X. Cheng, and D. Chen, "Insider Attacker Detection in Wireless Sensor Networks," in *Proc. IEEE INFOCOM 2007*, pp. 1937–1945.

[9] A. Meka and A. K. Singh, "Distributed Spatial Clustering in Sensor Networks," in *Proc. 10th International Conference on Extending Database Technology (EDBT 2006)*.

[10] A. Stetsko, L. Folkman, and V. Matya, "Neighbor-based Intrusion Detection for Wireless Sensor Networks," in *Proc. 2010 6th International Conference on Wireless and Mobile Communications (ICWMC 2010)* pp. 420–425.

[11] W-T. Su, K-M. Chang, and Y-H. Kuo, "eHIP: An Energy-Efficient Hybrid Intrusion Prohibition System for Cluster-based Wireless Sensor Networks," *Computer Networks*, vol. 51, pp. 1151–1168, 2007.

[12] M. Xie, S. Han, B. Tian, and S. Parvin, "Anomaly Detection in Wireless Sensor Networks: A Survey," *Journal of Network and Computer Applications*, vol. 34, pp. 1302–1325, 2011.

ON BANDWIDTH CAPABILITES OF MULTIPROCESSOR INTERCONNECTION NETWORKS

Dr. Sandeep Sharma

Department of Computer Science and Engineering, Guru Nanak Dev University, Amritsar,Punjab, India

Abstract - *In this paper the general characteristics and bandwidth capabilities of Multiprocessor Interconnection Networks have been analyzed. We have examined some popular FT(Four Tree)[8],MFT(Modified Four Tree)[2],THN(Theta Network)[13],NFT(New Four Tree)[4], IFT(Improved Four Tree)[5], IASN(Irregular Augmented Shuffle Network)[14] and IIASN(Improved Irregular Augmented Shuffle Netwrok) [3] networks which are irregular in nature[11]. The structural characteristics comparison of these networks is presented in terms of number of switches, links and stages etc. The bandwidth [15] and probability of acceptance of all the networks evaluated by using probabilistic approach [6] and their simulation results are compared.*

Keywords: *Multistage Interconnection Network, Permutation , Bandwidth , Probability of acceptance ,IIASN, Four Tree Network, NFT,THN,IFT*

1 Introduction

Parallel processing systems have a huge number of processors and an interconnection network to provide interprocess communication. One of the major design issues of these system to make these system fault tolerant[12], reliable and excellent in performance wise. Multistage interconnection networks (MINs) which are widely used in designing of multi processor architecture are attracting much concentration from parallel processing. The MINs are categorized as regular and irregular depending on the internal arrangement of switching elements SEs. In irregular the number of switches in each stage can vary. Most previous work has focused on evolution of the performance of regular MINs like as OMEGA, ESC, Multistage Cube, ASEN[11] and ABN[7] etc., but little work has been done for irregular class of Networks. In this paper we have considered some popular irregular networks which are multipath in nature. In this paper it is assumed that each processor is attached to an input port and an output port of considered MINs. However these results are also applicable to processor-memory interaction. The performance of all surveyed MINs is evaluated on the basis of bandwidth utilization and probability of acceptance of request which are coming from source processor to destination processor. The structural characteristics of surveyed MINs are also summarized in terms of number of switches, link and mux/demux used for the construction of them.

The rest of the paper is organized as follows. Section 2 discusses the structural characteristics of the surveyed MINs. Section 3 discuses the analytical model of probability of acceptance and bandwidth analysis of considered MINs. Finally conclusions have been given in Section 4.

2 Structural Characteristics of MINs

Table 1 summarizes the structural characteristics of surveyed MINs and it has been observed that all the networks which are used for inter processor communication are fault tolerant. One can choose the network on behalf of number of switches, number of links, multiplexers, demultiplexers and express links to reduce the cost. In later discussion it is summarized that the network with more switches and links have great failure rate in comparison to a network with lesser switches and links.

3 Probability of acceptance and Bandwidth

Probability of acceptance (POA) is defined as the fraction of incoming requests accepted by the output stage of network [7].It is the ratio of expected band-width (i.e. total no of successful requests) to the expected number of requests generated per cycle. Patel et al. suggested a probabilistic

approach[6] to evaluate these parameters with following assumptions:

- At each cycle, packets are generated at each source independently with probability p.
- Each packet is send with equal probability to any destination.
- The packet rate i.e the number of packets issued per cycle by a processing element , is p where p<=1
- Requests that are not accepted during a memory cycle are discarded.

For a system with N processors and M memory modules , if a processor generates a request with probability p in a cycle directed to each memory with equal probability then the bandwidth is given by Strecker et al. as :

$$BW = M(1-(1-p/M)^N) \quad (1)$$

Table 1: Summary of Structural Characteristics of MINs

MIN	Number of 2x2 Switches	Number of links/stage excluding express links	Number of switches with conjugate loops	Number of Stages	Type/ Number of mux	Type/ Number of demux	Redundant paths
FT	N/2	N(input and output stages) 2N+4 (interior)	N+2	Log_2N+1	2:1/N	1:2/N	Yes
MFT	N/2	N(input and output stages) 2N+4 (interior)	N+4	Log_2N+1	2:1/N	1:2 /N	Yes
THN	N+N/2	N(input and output stages) 2N (interior)	Zero	Log_2N	2:1/N	1:2 /N	Yes
NFT	N/2	N(input and output stages) 2N (interior)	N	Log_2N+1	2:1/N	1:2 /N	Yes
IFT	N/2	N(input and output stages) N+N/2 (interior)	N-N/4	Log_2N-1	4:1/N	1:2 /N	Yes
IASN	N/2	N(input and output stages) N+N/2 (interior)	N-N/4	Log_2N-1	4:1/N	1:2 /N	Yes
IIASN	N/2+2	N(input and output stages) N+N/4 interior	N/2	Log_2N-1	4:1/N	1:2 /N	Yes

Table 2 : Summary of Bandwidth Analysis of Various MINs when n=4

Probability of issuing a request	FT	MFT	THN	NFT	IFT	IASN	IIASN
0.1	1.2	1.4224	0.9621	1.4784	1.136	1.136	1.4928
0.2	2.1264	2.5536	1.8518	2.744	2.152	2.152	2.7888
0.3	3.0144	3.4624	2.6737	3.832	3.064	3.064	3.9152
0.4	3.8032	4.2032	3.4317	4.7712	3.8784	3.8784	4.896
0.5	4.5072	4.8112	4.1295	5.584	4.6096	4.6096	5.7504
0.6	5.1344	5.3152	4.7702	6.2912	5.2624	5.2624	6.496
0.7	5.696	5.736	5.3567	6.9072	5.848	5.848	7.1456
0.8	6.1968	6.0896	5.8913	7.4448	6.3696	6.3696	7.712
0.9	6.6432	6.3872	6.3763	7.9152	6.8352	6.8352	8.2064
1.0	7.0416	6.64	6.8135	8.3264	7.248	7.248	8.6368

Where p/M is the probability that a processor requests a particular memory module, $[1-(p/M)]^N$ is the probability that none of the N processors requests the memory module in a particular cycle.

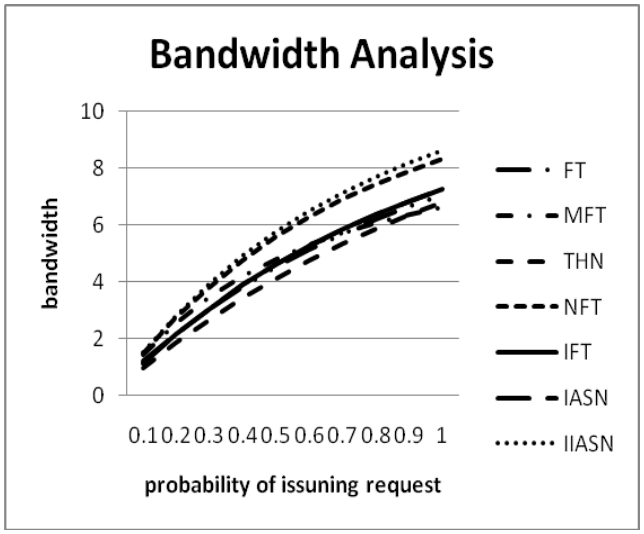


Figure 1: Bandwidth Analysis of various MINs

The table no. 2 shows that the bandwidth grows with the increase of probability of issuing the request. The graph in figure 1 shows that the probability of acceptance of various MINs remains constant when the probability of issuing the request is higher. It has been also observed that at the last stages when the probability of issuing of requests is 0.9 and 1.0 the difference between bandwidth in case of IIASN is 0.4304 quite less in comparison to 1.296 at very earlier states when probability of issuing the request was 0.1 and 0.2 which also lesser costlier[16] among all MINs.

4 Conclusions

The general characteristics in terms of structural characteristics has been analyzed one can choose the interconnection networks on the behalf of compared results. The performance in terms of bandwidth, probability of acceptance has been analyzed in this paper. The results for bandwidth show that the probability of maximum passing of request of IIASN network is higher than other networks. The bandwidth of IIASN gradually increases with the increase of probability of issuing a request.

5 References

[1] Algirdas Avizenis, "Towards Systematic Design of Fault-Tolerant Systems", IEEE Computer, Vol. 30, No. 4, April 1997, pp. 51-58

[2] Sandeep Sharma, P.K.Bansal, "A new fault tolerant Multistage Interconnection Network ", IEEE TENCON'02 , vol 1,2002, pp 347-350.

[3] Sandeep Sharma, K.S Kahlon, P.K.Bansal,Kawaljeet Singh, "Improved Irregular Augmented Shuffle Multistage Interconnection Network", International Journal of Computer Science and Security , vol 2 issue 3,2008, pp 28-33.

[4] Sandeep Sharma, P.K.Bansal, K.S. Kahlon, " On a class of Multistage Interconnection Network in parallel processing", International Journal of Computer Science and Network Security , vol 8 No. 5,May 2008, pp 287-291.

[5] Sandeep Sharma, K.S Kahlon, P.K.Bansal,Kawaljeet Singh, "Irregular class of Multistage Interconnection Network in parallel processing", Journal of Computer Science, vol 4(3) , 2008, pp 220-224.

[6] Patel,J.H.,"Performance of processor-memory interconnection for multiprocessors." IEEE Transaction on computers, 30, 1981, pp. 771-780

[7] P.K.Bansal, K. Singh, R.C. Joshi, and G.P. Siroha, "Fault tolerant Augmented Base Line Multistage Interconnection Networks", IEEE TENCON, 1991, pp. 205-208

[8] P.K.Bansal, K. Singh, and R.C. Joshi, "Routing and path length algorithm for a cost- effective four-tree multistage interconnection network", International Journal of Electronics, Vol. 73, 1992, pp.107-115,

[9] Subramanyam Arige , Prasad E.V and Nadumuni Reddy.C," Permuation Capabilyt and connectivity of Enhanced multistage interconnection Network(E-MIN)", International Conference on Advanced Computing and Communications, 2006. ADCOM 2006". 20-23 Dec. 2006, pp.8-11

[10] George, B.A., P.A. Dharma and H.J. Seigel, 1987. A survey and comparison of fault-tolerant multistage interconnection networks ,*Computer*, vol. 20, no. 6, Jun., 1987, pp. 14-27.

[11] James T. Blake and Kishor S. Trivdei, "Reliabilities of Two Fault Tolerent Multistage Interconnection Networks", IEEE Transactions on computers, 1988, pp. 300-305.

[12] Bataineh, S.M. and B.Y. Allosl, "Fault-tolerant multistage interconnection network", J.Telecomm. Syst., 17, 2001, pp. 455-472.

[13] J.Sengupta and P.K Bansal, "Performance evolution of a Fault Tolerant Irregular Networks ", Proceedings of IEEE TENCON'02, 2002, pp. 331-334

[14] Harsh Sadawatri , P.K Bansal, "Fault Tolerant Irregular augmented Shuffle Network", Proceedings of WSEAS International conference on Computer Engg. and Application, January 17-19, 2007, pp. 7-12

[15] Hu-Jun Wabg and Jian Li, "Bandwidth Analaysis for a class of Bus-Based Systems",The computer Journal Vol. 39,No.4,1996, pp 346-352.

[16] Sandeep Sharma, K.S Kahlon and P.K Bansal ,"Reliability and Path Length Analysis of Irregular Fault Tolerant Multistage Interconnection Network", ACM SIGARCH Computer Architecture. Vol. 37 ,2010 ,pp. 16-23.

QoS Guaranteed Handover Scheme for Global Roaming in Heterogeneous Proxy Mobile IPv6 Networks

Kwangsub Go[1] , Misun Kim[2], Kyujin Lee[3] and Youngsong Mun[3]

[1]KREONET Center, KISTI, Seoul, KOREA
[2]Administrative Management Division, Anti-Corruption and Civil rights Commission, Seoul, KOREA
[3]School of Computing, Soongsil University, Seoul, KOREA

Abstract - *Mobility and quality of service (QoS) are becoming the more important issues in wireless communications. The traditional Internet service is expanding into new access media and applications. Since wireless communication services are accompanied by frequent handovers at remote sites, scalable and fast handover has become a prerequisite for ubiquitous communication.*

In this paper, the differentiated service (Diffserv) model is deployed in heterogeneous proxy mobile IPv6 (PMIPv6) networks to satisfy the QoS guaranteed service and fast handover requirements. The operational procedures for QoS guaranteed global roaming are presented. In addition, QoS management and handover cost evaluation schemes based on a mobile host's movement scope are proposed. This paper analyzes the reduction in handover delay in a network-based localized mobility management framework. We propose and analyze a PMIPv6 optimized with a global mobile access gateway (G-MAG), which is a network-based entity, to further improve the handover performance in terms of handover delay while maintaining minimal signaling overhead in the air interface among converged heterogeneous wireless networks. The handover signaling procedures with host-based MIPv6 are compared with network-based proxy MIPv6 (PMIPv6) and fast PMIPv6 assisted by G-MAG to show how much handover delay reduction can be achieved. Analytical results show that the handover delay is significantly reduced.

Keywords: QoS, PMIPv6, handover, global roaming, cost evaluation.

1. Introduction

Traditional Internet service does not consider issues with host mobility and QoS such as transmission delay, packet loss ratio, and bandwidth. However, providing secure and seamless mobility and guaranteed QoS become more critical issues in wireless mobile services. Furthermore, the services that are now being used on mobile networks, including, Internet broadcasting, teleconferencing, interactive role playing, and telemedicine, tend to require a predefined QoS.

In the near future, wireless networks are going to fully integrate different wireless access technologies to enable their users to exploit advantages of these various technologies and satisfy the QoS requirements of new applications. Various wireless communication protocols developed for different purposes are integrating and converging to ubiquitous communications, which request QoS-guaranteed, fast, and seamless roaming services. The increasing demands for ubiquitous and mobile computing will require the integration of various wireless access technologies such as WLAN (wireless LAN), 3GPP (3rd generation partnership project), 3GPP2 (3rd generation partnership project 2), and IEEE 802.16. To visualize these access technologies, wireless networks have been converging to the Internet Protocol (IP). For instance, the mobile IPv4 (MIPv4) [1] and mobile IPv6 (MIPv6) [2] protocols were already standardized by IETF (Internet Engineering Task Force). In addition, the IEEE 802.16e amendment [3] enhances IEEE 802.16 with mobility support for users moving at vehicular speeds, and the WiMAX (Worldwide Interoperability for Microwave Access) Forum has adopted IP mobility [4]. An IP level mobility protocol is needed to enable the users of these future wireless networks to roam freely between various access networks. Global roaming, guaranteed QoS, and vertical handover will be prevalent in the near future.

The Internet protocol IPv4 has been slowly progressing toward IPv6, and during this period both protocols have become a part of the Internet service infrastructure. The existing wired core Internet architecture that is linked to wireless access networks is going to evolve into wireless Internet environments. A large number of different wireless access networks are linked using a multi-hop infrastructure, including notebook PCs, PDAs, and small sensors, and are being served actively. The MIP (mobile Internet protocol) has been developed to provide mobile Internet services based on host mobility. Although it is a stable and mature technology, there are some obstacles to overcome to be widely deployed. It is too heavy to be implemented on small mobile devices and complicated message exchange procedures must be handled by the mobile host itself. As a network-based localized mobility protocol, PMIPv6 has been proposed to overcome the problems with MIP's host-based mobility and long handover latency [5]. The goal of PMIPv6 is specifying a simple extension of MIPv6 that would support network-based mobility for IPv6 hosts, while reusing the signaling and features of MIPv6. As an alternative to MIP, PMIPv6 reuses the MIPv6 entities and concepts as much as possible, but the mobility management procedures are carried by the network devices. Moreover, the mobility infrastructure in the PMIPv6 domain can provide mobility to an MN operating in IPv4, IPv6, or dual mode, even if the transport network is not an IPv4 or IPv6 network. As a result, a new standard for supporting an IPv4 host and IPv4 network in PMIPv6 was released [6] and a new protocol for supporting MIPv6 hosts attached in PMIPv6 is actively being discussed[7]. PMIPv6 is

applicable to various networks but is inappropriate for global roaming applications.

To solve the long handover latency of MIPv6 and localized movement in PMIPv6, we propose a new scheme that provides appropriate QoS and global roaming. This scheme adopts a QoS management server and global MAG. It extends the localized movement scope of PMIPv6 to heterogeneous global networks such as different ISPs or international regions without considering the MN's location and movement. It also analyzes the host mobility costs based on the MN's movement scope.

The rest of this paper is organized as follows. Section 2 describes the related works for reducing handover cost and providing QoS schemes in MIP networks. In section 3, we describe the architecture and functionality of PMIPv6 networks deploying Diffserv. The mobility management procedures for handover cost evaluation and supporting differentiated services are discussed. Section 4 presents the analytical results for Diffserv and the handover cost based on the MN's movement. Conclusions and remarks are presented in section 5.

2. Related Works

In MIPv4, an MN is identified by its home address, regardless of its current point of attachment in the network, and the MN is associated with a care-of-address (CoA) when it is away from home [1]. This triangular routing causes significant delay that degrades the handover performance. It is important for the mobility management to support fast and seamless handover with negligible delay, which enables active services without disruption. Thus, improvements were made and incorporated in a newer version of MIP called mobile IPv6 (MIPv6) to overcome some of the drawbacks [2]. It has been discovered that mobility can be more efficiently handled if the mobility management is divided into global mobility management and localized mobility management. Extensions of MIPv6 such as hierarchical mobile IPv6 (HMIPv6) [8] and fast handover for mobile IPv6 (FMIPv6) [9] have been proposed by IETF for efficient localized mobility management. The main goal of these localized mobility management protocols is to reduce the handover delay by localizing the registration of an MN [8] so that seamless service continuity can be achieved during roaming across wireless networks. Handover latency is mainly the result of delays caused by the discovery, configuration, authentication, and binding update procedures associated with a mobility event. Most of the recently proposed mobility management schemes have been host based, that is, the MN is directly involved in mobility-related signaling. Studies on handover cost evaluation to reduce these delays have been performed in various MIPv6 networks [10-16]. Handover anticipation based on layer 2 (L2) trigger information is used to reduce the registration delay [10-14]. M. Lopez *et al.* proposed a proactive handover scheme [10] that only considers the predictive mode. Both the predictive and reactive modes are considered to optimize the handover delay in PMIPv6, but the performance was analyzed separately for each mode [11-12]. To enhance the handover performance, K. Lee [13] proposed a cross-layering mechanism combined with IEEE 802.16e networks. For a more accurate performance evaluation in FMIPv6, S. Ryu *et al.* proposed combining the two modes and considered the probability of predictive mode failure (PPMF), which is affected by the radius of a cell, velocity of a mobile host, and L2 triggering time [14]. To enhance the handover performance in PMIPv6, they proposed optimizing the authentication delay and predicting the optimized route in [15, 16], respectively. Although PMIPv6 performs better than the typical host-based MIPv6 and its extensions in terms of handover performance [11, 12], it has a long handover delay when an MN moves away from its local area.

Mobile Internet traffic has recently increased at an exponential rate, and the new class of applications increases the need for QoS guarantees, which pose big challenges for the current wireless communication environments. The gap between QoS provisioning and demand has been significantly enlarged. Integrated services (Intserv) [17] and Diffserv [18] were proposed to solve the QoS problem in IP networks [19, 20]. Most QoS-providing models in wired and wireless networks focused on the buffer scheduling mechanisms based on traffic types. Various QoS provisioning schemes have been proposed for mobile IP networks [21-24]. However, these studies did not consider the MN's movement. In this paper, we propose a new scheme that allows an MN to move wherever it wants while receiving a guaranteed QoS. A priority queue model is used to provide differentiated QoS in our model. Because the BE traffic has the lowest priority, the highest and medium priorities are allocated to the EF class and AF class, respectively. To enable a performance analysis of the priority queue, we use the M/G/1 [25] queuing model.

3. QoS Guaranteed Global Roaming Model in Heterogeneous PMIPv6 Networks

3.1. Basic Operation of PMIPv6

PMIPv6 reuses the infrastructure of existing MIPv6 protocols but adopts new entities such as a mobile access gateway (MAG) and local mobility agent (LMA) for network-based mobility management. The MAG monitors the MN's movement in access links and sends signaling messages to LMA instead of the MN. The LMA is an anchor point of the MN that assigns a home network prefix to the MN and performs the HA role in PMIPv6 domain. It also manages the MN's reachability state in the domain. Typically, the MAG functionalities are embedded in an access router and the LMA is stacked in the gateway. There is an IP tunnel for transferring signals and data packets between an MAG and LMA. When an MN is attached to the PMIPv6 domain, the MAG tries to acquire the MN's profile from the policy server. That profile contains the MN's ID and IP address configuration method for access link and may include the MN's home network address of IPv6. When the MAG acquires the profile of the MN, it

sends a proxy binding update message to register the MN's location to the LMA. After receiving the proxy binding update message, the LMA sends a proxy binding acknowledge message, including the MN's home prefix information, and creates a bi-directional tunnel to the MAG. It also manages a routing table for transferring data to the MN and maintains the MN's reachability. Then, the MN sets up its IP address through a router advertisement procedure to get information about its home network prefix and address configuration method. When an LMA receives external packets from the PMIPv6 domain for the MN, it forwards them through the tunnel to the MAG, which will eventually forward them to the MN.

3.2. QoS Guaranteed Handover Scheme for Global Roaming

PMIPv6 provides mobility without the participation of the host, but its mobility management scope is restricted within an LMA domain. However, the host needs inter-LMA and inter-Internet service provider (ISP) movement, as well as global roaming. In this section, we propose a roaming scheme which includes global roaming. Based on the MN's movement scope, which might be intra-LMA, inter-LMAs, or inter-ISP domains, we analyze the QoS guaranteed handover costs.

Fig. 1 illustrates the proposed network architecture for QoS-guaranteed, fast, global roaming in heterogeneous PMIPv6. In this model, we adopt new entities: a QoS agent (QA) to support differentiated service and global MAG (G-MAG) for fast handover when an MN moves between ISP domains. This scheme could provide QoS guaranteed services depending on a subscriber's service level agreement (SLA) in PMIPv6 networks. More details of the procedures for an MN's movement and SLA management are provided in [26, 27]. We assume that an ISP network is composed of various access infrastructures according to its QoS level and locality. An ISP has a global QoS agent (GQA or QA) that acts as a global QoS manager, like a bandwidth broker in a Diffserv network. Neighboring GQAs can communicate with each other to establish an inter-domain QoS association such as SLAs. Each subnet of an ISP may have its own local QoS agent (LQA or QA) and HA for QoS and mobility management within its local area. Each LQA can manage the resources within its subnet, and serves the MN with a service profile.

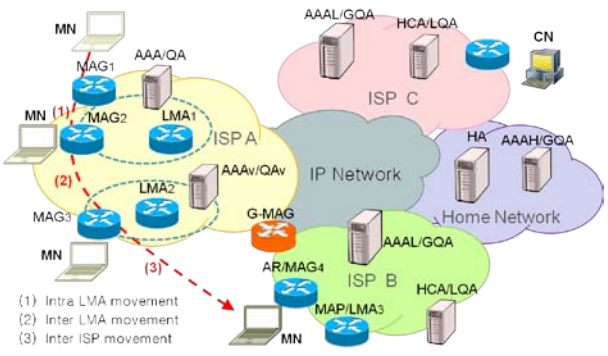


Fig. 1 **Network architecture for QoS-guaranteed, fast, global roaming.**

Three MN movement scope cases are shown in Fig. 1. In the first two cases, an MN moves around within the same ISP domain, while it moves into a neighboring ISP domain in the other case. In this figure, (1) indicates intra-LMA movement, which means that an MN attached to MAG1 moves into the new access point (AP) linked to MAG2, where both MAGs are managed by the same LMA1. Movement (2) represents inter-LMA movement. This means that the MN attached to MAG2 moves into MAG3, where the MAGs are managed by different LMAs, but both of them belong to the same ISP domain. Movement (3) shows inter-ISP movement, which means that an MN moves into another ISP domain's access point. This movement is called global roaming, which causes complicated procedures for handover. It causes long delays and QoS degradation. A global MAG (G-MAG) may solve this problem. It is geographically located on the border between ISP domains and used to connect the domains with a security association (SA). Thus, the G-MAG can manage the LMAs, AAAs, and QAs of both domains. In addition, it is able to transfer its profile and authentication information from one LMA to the other LMA during the pre-inter-domain handover. The G-MAG is able to estimate the MN's location and detect its movement by tracing the access point where the MN is attached. Thus, it can properly predict the point of the inter-domain handover execution. If the MN's inter-domain handover is imminent, the G-MAG performs a pre-inter-domain handover in advance of the inter-domain handover between the previous MAG and new MAG while the MN is still connected to the G-MAG.

3.3. Performance Analysis

This section describes a QoS-guaranteed handover cost analysis model that depends on an MN's movement scope. The handover cost of PMIPv6 C_{HO} can be expressed as the sum of movement detection latency T_{MD}, proxy binding update latency T_{PBND}, and router advertisement latency T_{RA}:

$$C_{HO} = T_{MD} + T_{PBND} + T_{RA}, \qquad T_{PBND} = T_{AUTH} + T_{QoS} + T_{CONF} \tag{1}$$

where T_{PBND} can be expressed as the sum of authentication delay T_{AUTH}, QoS management delay T_{QoS}, and address configuration delay T_{CONF}.

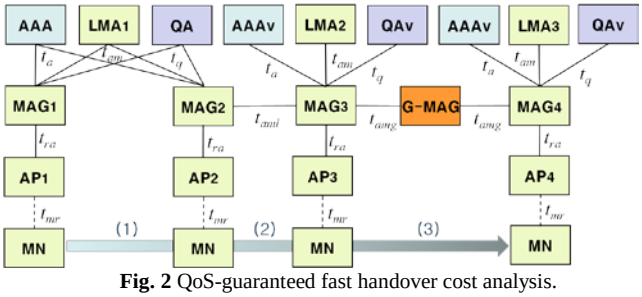


Fig. 2 QoS-guaranteed fast handover cost analysis.

Fig. 2 shows the QoS-guaranteed handover cost analysis model based on Fig. 1, which shows all three MN movement scenarios. The notations (1), (2), and (3) represent intra-LMA,

inter-LMA, and inter-ISP movements, respectively.

As shown in (1) of Fig. 1, when an MN moves to MAG2 from MAG1, MAG2 should register the MN to its LMA. Since the handover procedure is performed by the MN's movement detection, when an MN attaches to a new AP that is covered by a new MAG (MAG2), the AP sends L2 handover messages to the MAG2. This causes MN movement detection delay T_{MD}. When MAG2 detects the MN's movement, it sends authentication and QoS profile request messages for the MN to the AAA and QA servers, respectively. After receiving the authentication and QoS acknowledgement messages for the above queries, MAG2 sends a proxy binding update (PBU) message to its LMA. If the LMA can allocate the home network prefix to the MN, it sends a proxy binding acknowledgement (PBA) message to MAG2. MAG2 sends a router advertisement (RA) message to the MN when it receives a PBA message from the LMA. The intra-LMA handover in PMIPv6 is now completed. In this case, both MAGs (MAG1 and MAG2) are in the same region of LMA1, and the MN's registration procedures follow the conventional PMIPv6 protocol [5]. The QoS-guaranteed handover cost of intra-LMA movement in PMIPv6 network $C_{P\_HO}^{INTRA\_LMA}$ can be described in expression (2):

$$\begin{aligned}
C_{P_HO}^{INTRA_LMA} &= T_{P_MD}^{INTRA_LMA} + T_{P_PBNB}^{INTRA_LMA} + T_{P_RA}^{INTRA_LMA} \\
&= 2(t_{ra} + t_a + t_q + t_{am}) + t_{mr}, \\
T_{P_MD}^{INTRA_LMA} &= t_{ra},\ T_{P_RA}^{INTRA_LMA} = t_{ra} + t_{mr}, \\
T_{P_PBND}^{INTRA_LMA} &= T_{P_AUTH}^{INTRA_LMA} + T_{P_QoS}^{INTRA_LMA} + T_{P_CONF}^{INTRA_LMA} \\
&= 2(t_a + t_q + t_{am}), \\
T_{P_AUTH}^{INTRA_LMA} &= 2t_a,\ T_{P_QoS}^{INTRA_LMA} = 2t_q, \\
T_{P_CONF}^{INTRA_LMA} &= 2t_{am},
\end{aligned} \tag{2}$$

where t_{ra} is the movement detection delay, t_{mr} and t_{am} are wireless and wired delays, and t_a and t_q are authentication and QoS setup delays, respectively.

An inter-LMA handover cost analysis is shown in (2) of Fig. 2. Since PMIPv6 is designed to manage local area movement, if an MN is away from its current LMA domain, registration procedures for the MN should be carried out through its home network. This causes a long delay and QoS degradation. To reduce the handover delay through the MN's home network, we propose the use of a pre-handover mechanism before detaching the current MAG (MAG2). When an MN receives a new router advertisement message with a network prefix that is different from its current one, it sends a request to its current MAG (MAG2) to send its profile to the new MAG (MAG3) before detaching from MAG2.

In this message, the MN's profile is included for movement management without link disruption. Since MAG3 receives the MN's profile from MAG2, it does not need to request authentication and QoS properties from its home network entities. Thus, we can significantly reduce the handover cost. When an MN moves to a new MAG that belongs to a new LMA domain, the proposed fast handover cost is represented by equation (3):

$$\begin{aligned}
C_{fP_HO}^{INTER_LMA} &= T_{fP_MD}^{INTER_LMA} + T_{fP_PBND}^{INTER_LMA} + T_{fP_RA}^{INTER_LMA} \\
&= 2(t_{ra} + t_a + t_q + t_{am}) + t_{mr} + 6t_{aml}, \\
T_{fP_MD}^{INTER_LMA} &= t_{ra},\ T_{fP_RA}^{INTER_LMA} = t_{ra} + t_{mr}, \\
T_{fP_PBND}^{INTER_LMA} &= T_{fP_AUTH}^{INTER_LMA} + T_{fP_QoS}^{INTER_LMA} + T_{fP_CONF}^{INTER_LMA} + T_{fP_PRE_REG}^{INTER_LMA} \\
T_{fP_AUTH}^{INTER_LMA} &= 2(t_a + t_{aml}),\ T_{fP_QoS}^{INTER_LMA} = 2(t_q + t_{aml}), \\
T_{fP_CONF}^{INTRA_LMA} &= 2(t_{am} + t_{aml}), \\
T_{fP_PRE_REG}^{INTER_LMA} &= t_{mr} + t_{ra},
\end{aligned} \tag{3}$$

where t_{aml} is the delay for the MN's profile transfer between the two MAGs and pre-handover delay $T_{fP\_PRE\_REG}^{INTER\_LMA}$, which occurs before the MN's movement to support for the MN's movement between LMAs, can be neglected. An inter-ISP handover cost analysis is shown in (3) of Fig. 2. If it has to be executed in the conventional PMIPv6 schemes, the MN suffers from a long service disruption because of the home registration of the MN. When the MN moves from MAG3 to MAG4, the inter-ISP handover latency becomes too long to support seamless service continuity. To reduce this long inter-ISP handover latency, we propose a new scheme that adopts a network entity called G-MAG, as shown in Fig. 1. The G-MAG is located between ISP domains and is connected to both domains with a security association (SA). Upon predicting an imminent inter-domain handover for the MN, it performs the inter-domain handover while the MN is connected to the previous MAG (MAG3) in the previous domain (ISP A). Since the G-MAG has dual connections between ISP domains, it can gather and manage the LMA information of both domains. It can also transfer the MN's profile and authentication information from one LMA to the other LMA during the pre-inter-domain handover. The G-MAG is able to estimate the MN's location and movement direction by tracing the AP where the MN attaches. Consequently, the G-MAG can properly predict the inter-domain handover execution point. Thus, because the new MAG (MAG4) receives the MN's profile from the G-MAG, it does not need to request authentication and setup QoS properties from its home network.

Therefore, the proposed scheme can reduce the inter-domain handover latency by avoiding the signaling to the MN's home network. When an MN moves to a new MAG that belongs to a new ISP domain, the handover cost of the proposed scheme is represented by equation (4):

$$\begin{aligned}
C_{fP_HO}^{INTER_ISP} &= T_{fP_MD}^{NTER_ISP} + T_{fP_PBND}^{NTER_ISP} + T_{fP_RA}^{NTER_ISP} \\
&= 2(t_{ra} + t_{am} + t_a + t_q) + 6t_{amg} + t_{mr}, \\
T_{fP_MD}^{INTER_ISP} &= t_{ra},\ T_{fP_RA}^{NTER_ISP} = t_{ra} + t_{mr}, \\
T_{fP_PBND}^{INTER_ISP} &= T_{fP_AUTH}^{INTER_ISP} + T_{fP_QoS}^{INTER_ISP} + T_{fP_CONF}^{INTER_ISP}, \\
T_{fP_AUTH}^{INTER_ISP} &= 2(t_{amg} + t_a),\ T_{fP_QoS}^{INTER_ISP} = 2(t_{amg} + t_q), \\
T_{fP_CONF}^{INTER_ISP} &= 2(t_{am} + t_{amg}),
\end{aligned} \tag{4}$$

where t_{amg} is the delay between MAGs that belong to

different ISP domains.

To show our model's handover cost effectiveness, we present a handover cost analysis model for the conventional PMPv6 in Fig. 3. It shows a cost evaluation for inter-LMA and inter-ISP handovers in the conventional PMIPv6. As can be seen in this figure, the intra-LMA handover cost is identical to (1) of Fig. 2. Thus, the intra-LMA handover cost in the conventional PMIPv6 is the same as expression (1). However, when an MN moves to LMA2 from its previous LMA (LMA1), the new MAG (MAG3) and LMA2 have no information about the MN if these events occur in conventional PMIPv6 networks. The delay associated with acquiring the MN's profile, authentication, and QoS information through the MN's home network should be taken into account, in addition to the handover costs for inter-LMA and inter-ISP movements in the conventional PMIPv6.

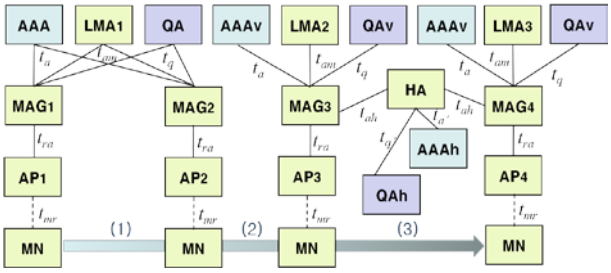


Fig. 3 **Handover cost analysis in conventional PMIPv6.**

In Fig.3, (2) and (3) represent the inter-LMA and inter-ISP handover cost evaluations. When an MN moves into a new LMA or new ISP domain that does not belong to its previous network, there are no benefits to PMIPv6 because it was developed as a local movement management scheme. Thus, all of the mobility procedures have to be carried out like in MIPv6 networks. When a new MAG such as MAG2 or MAG3 detects an MN's movement, it has to acquire the MN's profile, authentication, and QoS information through the MN's home network. After receiving the MN's profile from its home network, the new MAG sends a PBU message to its LMA (LMA2 or LMA3). After receiving the PBA message from the new LMA, the new MAG sends an RA message to the MN, and the inter-LMA or inter-ISP handover is completed. The handover costs for these two cases are the same because the handover procedures for inter-LMA and inter-ISP movements are identical. The handover cost is expressed by (5):

$$
\begin{aligned}
C_{cP\_HO}^{INTER\_LMA,ISP} &= T_{cP\_MD}^{INTER\_LMA,ISP} + T_{cP\_PBNB}^{INTER\_LMA,ISP} + T_{cP\_RA}^{INTER\_LMA,ISP} \\
&= 2(t_{ra} + t_a + t_q + t_{a'} + t_{q'} + t_{am}) + 6t_{ah} + t_{mr}, \\
T_{cP\_MD}^{INTER\_LMA,ISP} &= t_{ra}, \quad T_{cP\_RA}^{INTER\_LMA,ISP} = t_{ra} + t_{mr}, \\
T_{cP\_PBND}^{INTER\_LMA,ISP} &= T_{cP\_AUTH}^{INTER\_LMA,ISP} + T_{cP\_QoS}^{INTER\_LMA,ISP} + T_{cP\_CONF}^{IINTER\_LMA,ISP} \\
&= 2(t_a + t_q + t_{a'} + t_{q'} + t_{am}) + 6t_{ah}, \\
T_{cP\_AUTH}^{INTER\_LMA,ISP} &= 2(t_a + t_{ah} + t_{a'}), \; T_{cP\_QoS}^{INTER\_LMA,ISP} = 2(t_q + t_{ah} + t_{q'}), \\
T_{cP\_CONF}^{INTER\_LMA,ISP} &= 2(t_{am} + t_{ah}),
\end{aligned}
\tag{5}
$$

where t_{ah}, $t_{a'}$, and $t_{q'}$ are the delays for the MN's registration, authentication, and QoS management through its home network, respectively.

4. Cost Evaluation Results

In this section, we compare the handover costs of the conventional scheme and our proposed models in heterogeneous PMIPv6 networks by service types and the MN's movement scope. The parameters used in this handover cost analysis are defined as follows. Let's assume that a CN generates data packets destined for an MN at mean rate α and an MN moves from one subnet to the other at mean rate β. The packet to Mobility Ratio (PMR), $\rho = \alpha/\beta$, is defined as the mean number of packets received by an MN from a CN per movement. The parameters l_c and l_d are defined as the average length of a control packet and data packet, respectively. Then, their ratio, l, can be defined as l_c/l_d. The cost of transmitting a control packet is given by the distance between a sender and a receiver. The cost of transmitting a data packet is l0 times greater than the average cost of processing a control packet at any host (and forwarding data packets at an HA). The average delay time and packet loss probability are very important factors for QoS. In Diffserv networks, ingress edges classify the traffic by SLA. We assume three service levels: EF, AF, and BE. We also assume that an access router accommodates different types of data packets from numerous connections. To serve these three types of traffic, we use 3 buffers in the output buffer module of each node. In this paper, each node has an M/G/1 queuing model for evaluating the performance of prioritized packets. Packets are summed to arrive in the queue according to a Poisson process with mean rates λ_1, λ_2, and λ_3 for EF, AF, and BE packets, respectively. The service times for packets from each traffic class follow exponential distributions, with mean rates of $1/\mu_1$, $1/\mu_2$, and $1/\mu_3$ for EF, AF, and BE packets, respectively. The mean offered load of the EF, AF, and BE packets in the buffer are $\rho_1=\lambda_1/\mu_1$, $\rho_2=\lambda_2/\mu_2$, and $\rho_3=\lambda_3/\mu_3$, respectively. The packet scheduling at the buffer module is as follows. First, the server visits an EF buffer. If packets exist in the EF buffer, it serves them until the buffer is empty. Otherwise, the server visits an AF buffer, and serves the packets in that buffer. After the service is finished for the AF buffer, BE packets are served.

To compare our QoS-guaranteed model with the existing extensions of MIPv6, we use the wired and wireless experimental results in [11, 12]. The values are $t_{mr} = t_{am} = t_{hc} = 10$ *ms*, $t_a = t_q = t_{a'} = t_{q'} = t_{ah} = t_{ac} = 20$ *ms*, $t_{ra} = 2$ *ms*, $t_{amg} = 6$ *ms*, and $t_{aml} = 4$ *ms*. The control packet size and data packet size are assumed to be 100 bytes and 1,024 bytes, respectively, and the buffer size, K, is assumed to be 100. The traffic arrival rates are $\lambda_1 = 0.5$, $\lambda_2 = 0.3$, and $\lambda_3 = 0.2$ under a work-load of 0.5.

Figs. 4, 5, and 6 show the handover costs according to service type and MN movement scope. Fig. 4 shows the intra-

LMA handover costs of MIPv6 and PMIPv6. In this case, we represent the cost of handover from the conventional PMIPv6 (cPMIPv6) and proposed fast PMIPv6 (fPMIPv6) as PMIPv6 because when an MN moves around in an LMA domain, the handover procedures are identical, and thus the handover costs are the same. As can be seen, the difference between the handover costs of MIPv6 and PMIPv6 is significant because of the MN's CoA registration in MIPv6 networks. It also shows the service types are a significant element influencing the handover costs. As shown in Fig. 4, the main factor for the handover cost is the service policy. PMIPv6 shows very good performance compared to MIPv6. In the PMIPv6 network, the handover costs for the EF, AF, and BE services are 9.64 times smaller than MIPv6's for all of the traffic.

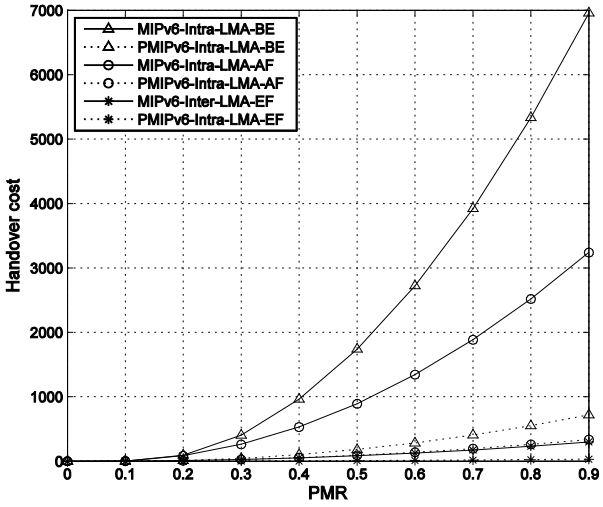


Fig. 4 Intra-LMA handover costs in PMIPv6 networks.

The inter-LMA and inter-ISP handover costs in PMIPv6 are shown in Fig. 5 and Fig. 6, respectively. In these figures, we omit the MIPv6 costs because the results are the same as those of Fig. 4. Thus, we compare the results of the conventional PMIPv6 and proposed fast PMIPv6 schemes. When an MN moves between LMAs, the handover costs of fPMIPv6 are significantly lower than those of cPMIPv6. This is because when an MN moves to a different LMA in a cPMIPv6 network, additional handover procedures to acquire the MN's profile are needed, as in MIPv6. All of the fPMIPv6 service types show better performance than the AF service of cPMIPv6. The handover costs of fMIPv6 are 2.15 times smaller than those of cPMIPv6 for all of the traffic. This shows that our scheme is effective for inter-LMA movement.

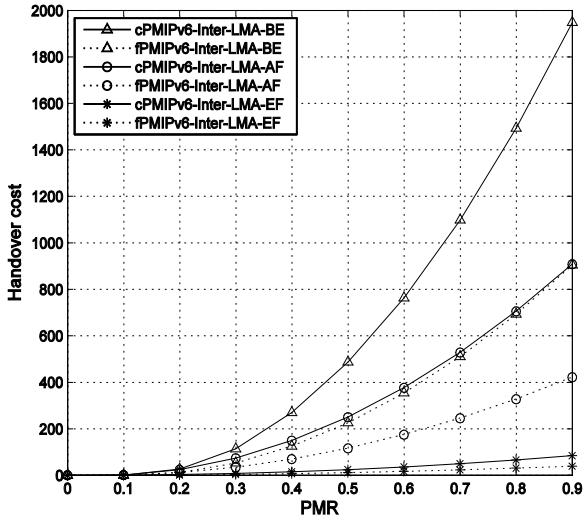


Fig. 5 Inter-LMA handover costs in PMIPv6 networks.

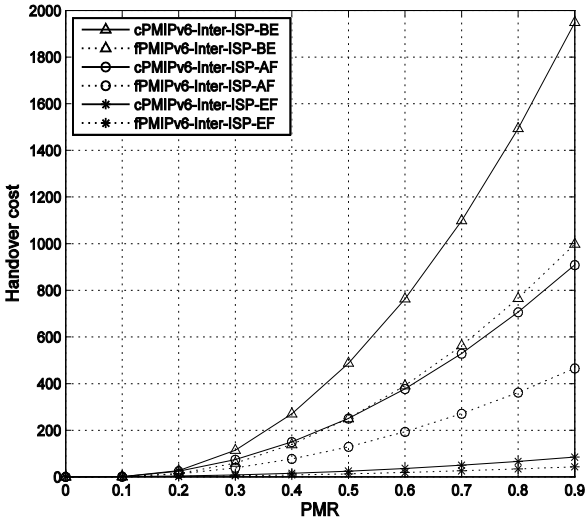


Fig. 6 Inter-ISP handover costs.

Fig. 6 shows the inter-ISP handover costs. In this case, we adopt a G-MAG to eliminate the MN authentication process and QoS setup delay involving its home network. The handover costs are shown for BE, AF, and EF traffic in the conventional and fast PMIPv6. Our scheme reduces the handover cost by 1.65 times compared to cPMIPv6. The results show that fPMIPv6 is effective under global roaming circumstances and could be adopted in future global networks that require an MN that is fast and QoS-guaranteed roaming.

5. Conclusions

Various applications and portable communication devices demand various QoS and mobility levels under different circumstances. To provide a guaranteed QoS and fast mobility, we propose procedures for mobility and QoS management in heterogeneous PMIPv6 networks. When Diffserv is deployed in a mobile IP network, various problems are encountered. The most significant problem is related to the acquisition of the service profile for the MN when it moves into a different ISP domain. Since the first-hop router does not have any information about the newly attached MN, a guaranteed QoS could not be provided. To solve this problem, we present procedures for acquiring the MN's service profile and additional information based on the MN's movement scope. Through a guaranteed-QoS handover cost analysis and evaluation, we reduce the handover cost by 9.64 times compare to MIPv6 for intra-LMA movement. Moreover, our scheme shows handover cost efficiencies for inter-LMA and inter-ISP movements that are 2.15 and 1.64 times better than the conventional PMIPv6, respectively. Consequently, our scheme is effective under global roaming circumstances and could be adopted in future global networks that require a fast speed, guaranteed QoS, and roaming. In future work, we will refine the message types and registration procedures to manage the MN's mobility more effectively and reduce the total communication cost. An authentication mechanism for seamless handover will also be considered to construct a service that is more efficient and secure, with a guaranteed QoS in heterogeneous PMIPv6 networks.

6. References

[1] C. Perkins, "IP mobility support for IPv4," IETF RFC 3344, Aug. 2002.

[2] D. Johnson, C. Perkins, and J. Arkko, "Mobility Support in IPv6," IETF RFC 3775, June 2004.

[3] "Part 16: air interface for fixed and mobile broadband wireless access systems-amendment 2: physical and medium access control layers for combined fixed and mobile operation in licensed bands," IEEE Std 802.16e-2005, Feb. 2006.

[4] "Stage 2: architecture tenets, reference model and reference points. Part 2," WiMAX Forum Network Architecture, Feb. 2009.

[5] S. Gundavelli, K. Leung, V. Devarapalli, K. Chowdhury, and B. Patil, "Proxy mobile IPv6," IETF RFC 5213, Aug. 2008.

[6] R. Wakikawa and S. Gundavelli, "IPv4 Support for Proxy Mobile IPv6," IETF RFC 5844, May 2010.

[7] G. Giaretta, "Interaction between PMIPv6 and MIPv6: Scenarios and Related Issues," draft-ietf-netlmm-mip-interactions-06, May 2010.

[8] H. Soliman, C. Castellucia, K. Malki, and L. Bellier, "Hierarchical Mobile IPv6 Mobility Management (HMIPv6)," IETF RFC 4140, Aug. 2005.

[9] R. Koodli, "Fast Handovers for Mobile IPv6," IETF RFC 4068, July 2005.

[10] A. Dutta, S. Das·, D. Famolari, Y. Ohba, K. Taniuchi, V. Fajardo, R. M. Lopez, T. Kodama, and H. Schulzrinne, "Seamless proactive handover across heterogeneous access networks," Wireless Personal Communication, Vol. 43, Issue 3, pp. 837-835, Nov. 2007.

[11] C. Makaya and S. Pierre, "An Analytical Framework for Performance Evaluation of IPv6-Based Mobility Management Protocols," IEEE Trans. on wireless communications, Vol. 7, No. 3, pp. 972-983, March 2008.

[12] K. Kong, W. Lee, Y. Han, M. Shin, and H. You, "Mobility management for All-IP mobile networks: mobile IPv6 vs. proxy mobile IPv6," IEEE Wireless Communications, vol. 15, no. 2, pp. 36–45, 2008.

[13] K. Lee and Y. Mun, "Enhanced Cross-Layering Mobile IPv6 Fast Handover over IEEE 802.16e Network," will appear in IEICE Transaction on Communications.

[14] S. Ryu, K. Lee, and Y. Mun, "Optimized fast handover scheme in Mobile IPv6 networks to support mobile users for cloud computing," Journal of Supercomputing, published in Online First, DOI 10.1007/s11227-010-0459-2, June 2010.

[15] L. Magagula and H. A. Chan, "Optimized handover delay in Proxy Mobile IPv6 using IEEE802.21 MIH Services," Military Communications Conference, 2008. MILCOM 2008. IEEE pp. 1-7, Nov. 2008.

[16] J. Na, S. Ryu, K. Lee, and Y. Mun "Enhanced PMIPv6 Route Optimization Handover Using PFMIPv6," IEICE Trans. Commun. Vol. E93-B, No. 11, pp. 3144-3147, Nov. 2010.

[17] R. Braden, D. Clark, and S. Shenker, "Integrated Services in the Internet Architecture: an Overview," IETF RFC 1643, June 1994.

[18] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. "An Architecture for Differentiated Services," IETF RFC 2475, Dec. 1998.

[19] M. May, S. Blot, A. Jean-Marie, and C. Diot, "Simple Performance Models of Differentiated Services Schemes for the Internet," Infocom'99, vol. 3, pp. 1385-1394, March 1999.

[20] S. Bakiras and V. O. K. Li, "Efficient Resource Management for End-to-End QoS Guarantees in Diffserv Networks," 2002 IEEE International Conference on Communications, Vol. 2, pp. 1220-1224, April 2002.

[21] K. M. Yusof and N. Fisal, "Provisioning QoS in Differentiated Service Domain for MIPv6," 4th National Conference on Telecommunication Technology 2003, pp. 98-101, January 2003.

[22] R. Jain, T. Raleigh, C. Graff, and M. Reschinsky, "Mobile Internet Access and QoS Guarantees using Mobile IP and RSVP with Location Registers," ICC98, pp. 1690-1695, June 1998.

[23] A. Mahmoodian and G. Haring, "Mobile RSVP with dynamic Resource Sharing," IEEE Wireless Communications and Networking Conference 2000, Vol. 2, pp. 896-901, September 2000.

[24] R. Chakravorty, M. D'Arienzo, I. Pratt, and J. Crowcroft. "A Framework for Dynamic SLA based QoS Control for UMTS," IEEE Wireless Communications, Special Issue on Merging IP and Wireless Networks, Vol. 10, No. 5, pp. 30-37, October 2003.

[25] L. Kleinrock, Queueing Systems, vol.1. J. Wiley & sons, 1975.

[26] M. Kim, S. Park, and Y. Mun, "QoS Guaranteed Service Model in Mobile IPv6," SAM'03, Las Vegas, U.S., vol. 2, pp. 502-507, June 2003.

[27] M. Kim and Y. Mun, "Cost Evaluation of Differentiated QoS Model in Mobile IPv6 Networks," ICCSA 2006, Glasgow, U.K., vol. 2, pp. 502-507, May 2006.

Constant Time Collision-Free Path Computation on Reconfigurable Mesh

Hatem M. El-Boghdadi
Computer Engineering Department, Faculty of Engineering,
Cairo University, Giza, Egypt
helboghdadi@eng.cu.edu.eg

Abstract - *The reconfigurable mesh (R-Mesh) was shown to be a very powerful model capable of extremely fast solutions to many problems. R-Mesh has a wide range of applications such as arithmetic problems, image processing and robotics. The 2D R-Mesh was shown to be able to solve the path planning problem very fast.*

In this paper, we propose an algorithm to compute a collision-free path between a source and a destination in an environment with the existence of obstacles. Independent of the number of obstacles, k, the proposed algorithm runs in constant time and requires O(log ^{2}N) pre-processing time where N is the size of the R-Mesh. This is in contrast to the previous work that requires O(k) time with the same pre-processing time. We then make a modification to the obtained path to enhance its length. This enhancement also requires constant time.

Keywords-*parallel algorithms, reconfigurable mesh, path planning*

1 Introduction

Reconfiguration is a very powerful computing paradigm, capable of extremely fast solutions to many problems. Models such as the reconfigurable mesh (R-Mesh) [9] (shown in Figure 1) have the ability to change the interconnection between processors at every step of the computation to allow efficient communication as well as performing computation faster than conventional non-reconfigurable models. A 2D R-Mesh is an array of processors with fixed external connections between each two neighboring processors. Also it has dynamically reconfigurable internal connections within each processor. This allows altering the interconnection among processors very fast, possibly at each step of the computation.

Robot motion planning algorithms aim to plan a path for a moving robot from a source location to a target location. The environment that the robot navigates through may have obstacles. The main target for the robot is to maneuver without colliding with obstacles if any. In general, the existent obstacles may be static or dynamic obstacles; *i.e.* moving obstacles. Restrictions on the robot movement from source to destination may take into account a number of factors. These include the robot shape and type of movement; for example translational or rotational. To simply handle the path planning problem, the image of robot and obstacles are digitized and stored in the R-Mesh, with one processor holding one pixel of the image. The R-Mesh was shown to handle these kinds of problems very fast. Also some techniques use first an algorithm to generate the *configuration space*. The configuration space is a slightly different image for the robot and the obstacles than the original image. It can be obtained by expanding the obstacles based on the shape of the robot and its movement. This way, the robot that is not point-like can be converted to a point-like robot. This greatly simplifies the design and analysis of the algorithm.

Since path planning problem is computationally intensive, many parallel algorithms have been proposed using various algorithmic models and assumed different robot shapes. These models include R-Mesh, hypercube computers, etc.

Tzionas *et al*. assumed a diamond-shaped robot and presented a parallel algorithm for collision-free path planning [12]. Jenq and Li [4] used hypercube to compute the configuration space optimally. The algorithm was shown to be optimal where it requires O(log n) time for an $n \times n$ image by using $n \times n$ Mesh of processors. D. Wang [14] proposed efficient algorithms for solving the reachability problem in one dimensional space. Dehne *et al*. [3] presented a systolic algorithm for computing the configuration space for obstacles in a plane for a rectilinear convex robot. The algorithm takes O(n) time for an $n \times n$ image on an $n \times n$ mesh of processors.

H.C. Lee [7] studied the maze-routing problem and it was shown that the R-Mesh is suitable for developing efficient and fast algorithms to solve the maze-routing problem. The maze-routing problem has its application in the path planning problem.

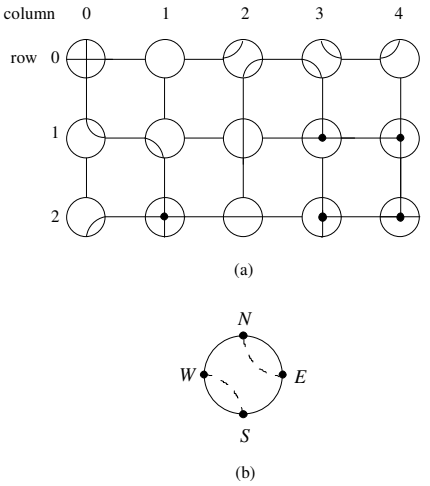


Fig.1 (a) Example of buses in a 3 × 5 R-Mesh (b) Processor ports

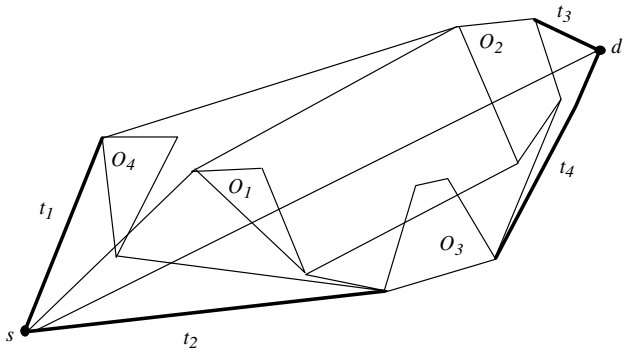


Fig.2 Illustration of the linear time algorithm [15]

Recently, D. Wang [15], used the 2D R-Mesh to compute a collision-free path. The obstacles are assumed to be disjoint convex or concave polygons. The algorithm starts by drawing a line from the source location to the destination location. Then, it finds the obstacles that intersect with that line if any. If more than one obstacle is found, then the algorithm combines these obstacles into one big obstacle. If only one obstacle is found or using the combined obstacle, the algorithm finds the tangent lines to that obstacle and finds if new obstacles intersect with these tangent lines. The mentioned operation is repeated till the four tangent lines to the combined obstacle from the source and destination do not intersect with any obstacles. At the end, there will be two possible collision-free paths from the source to destination. This operation requires all obstacles to be convex or concave polygons. The algorithm runs in O(k) time with preprocessing O(log ^2N) time. Figure 2 shows the two possible paths from the source s to the destination d for the given environment. In the figure, there are four obstacles labeled O_1, O_2, O_3 and O_4. The obstacles are combined into one obstacle, then the tangents t_1, t_2, t_3 and t_4 from the source and the destination are drawn to this combined obstacle. Thus there are two possible paths between s and d. Intuitively, the work in [15] finds a path that goes around the obstacles in the area between the source and the destination. With an O(log^2N) preprocessing time for the given obstacle image, the algorithm uses O(k) time to compute a path from a source to a destination while avoiding all obstacles in the environment, where N is the total number of processors (pixels) and k is the number of obstacles.

In this paper, we consider the R-Mesh for computing a collision-free path between a source and a destination in the presence of obstacles. Independent of the number of obstacles, k, the algorithm runs in constant time and requires O(log ^2N) pre-processing time where N is the size of the R-Mesh. This is in contrast to the work of Wang [15] that requires O(k) time with the same pre-processing time. The main idea is to start with a straight line between the source s and the destination d. We denote this line segment $\overline{sd}$. The path is generated such that it follows the straight line segment $\overline{sd}$ with going around each obstacle that intersects with this path. The algorithm generates 2^m possible paths where m is the number of obstacles that intersects with $\overline{sd}$, $1 \le m \le k$. Then an enhancement to the generated path is proposed. This enhancement tends to decrease the length of the generated path.

The next section presents some preliminaries and definitions. Section 3 describes the preprocessing operations that are applied before running the proposed algorithm. In Section 4, we describe the proposed algorithm and its time analysis. Section 5 presents an enhancement to the algorithm that could reduce the length of the generated path. In Section 6, we summarize our results and make some concluding remarks.

2 Preliminaries

In this section we introduce some preliminaries and definitions that are used through out this paper.

2.1 Reconfigurable Mesh

An $R \times C$ *reconfigurable mesh* or *R-Mesh* [13] consists of an R-row, C-column array of processors connected by an underlying mesh (see Figure 1). Number the rows (resp., columns) 0,1..., R-1 (resp., 0,1..., C-1) as shown in Figure 1. Each processor has four ports (called North, South, East, and West ports in the obvious manner, and abbreviated N, S, E, and W (see Figure 1(b)).

Each processor can independently partition its ports to connect certain ports together leaving other ports unconnected. For example, in Figure 1(a) the top left processor connects its N port to its S port, and its E port to its W port. The corresponding partition is denoted by $\{\overline{NS}, \overline{EW}\}$.

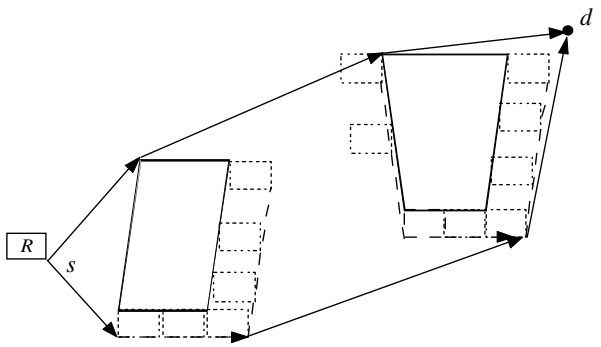


Fig.3 Expanding the original obstacles and reducing the robot to a point *s*

Figure 1(a) shows the fifteen possible port partitions of the R-Mesh.

An assignment of a port partition to each R-Mesh processor is called a *configuration*. Figure 1(a) shows one possible configuration. The port partitions along with the underlying mesh connections between neighboring processors form buses connecting processors. Figure 1 shows buses connecting sets of processors. An assumption central to all traditional R-Mesh algorithms is that buses have constant delay, regardless of the number of processors they span which enables us to design very fast algorithms. An R-Mesh making this assumption is called a unit-cost R-Mesh. In this paper we consider the unit-cost R-Mesh.

At each step of an R-Mesh algorithm, a processor could perform the following actions: (1) configure (partition) its ports, (2) read from or write to its ports, and (3) perform a local computation. An R-Mesh could permit concurrent reads and writes. If more than one processor is allowed to write to a bus at the same time, then the R-Mesh has concurrent write ability and the concurrent write rules [13] are used to resolve the values written to the bus. In this paper, we consider exclusive read exclusive write (EREW) R-Mesh.

2.2 Configuration Space

Usually the robot that navigates from a source to a destination has a certain shape and size. To be able to find a path without colliding with the existing obstacles, we need to take into account the shape and the size of the robot. One way to simplify handling the robot shape and size is to create what is called a *configuration space*. Instead of dealing with the robot as an object with a certain shape and size, we deal with it as a point and try to find a path from the source to destination. The configuration space is a slightly different image for the robot and the obstacles than the original image. It can be obtained by expanding the size of the obstacles that account for the shape and size of the robot. Thus, the robot can be reduced to a point. Many algorithms have been proposed to compute the configuration space for different robot shape and size [6]. For example if the robot is of polygon shape, the configuration space can be computed using the R-Mesh in O(1) time. In this paper we deal with the configuration space directly; i.e. we assume it has been already computed. Figure 3 shows the expanded obstacles for a rectangular robot.

3 Pre-processing Operations

The preprocessing operation, convexaziation, handles all the obstacles in the environment. The process builds a convex hull for each obstacle and enumerate the extreme points of each convex. Figure 4 shows an obstacle after applying the operation of convexaziation. The target is to identify the extreme points of the convex hull of the given polygon *H*. The figure shows the polygon after enumerating its extreme points. The work of Miller *et al*. [9] performs this operation on all the obstacles in the environment in $O(log^2 N)$ time where N is the size of the R-Mesh. After enumerating the extreme points, each extreme point processor has a flag to identify it as extreme point. Each extreme point processor has the following information (1) The ID number of the polygon this extreme point belongs to. (2) The number of the extreme point in the polygon. (3) The locations of the previous and the following extreme points. In other words, each extreme point contains segments information associated with it.

We assume that after convexization, *s* and *d* are not covered by any convex polygon. If that is not the case, a constant time operation treatment discussed later in section 6 reduces the problem to the assumed situation.

4 Constant Time Algorithm

In this section we describe a constant time algorithm to solve the path planning problem on the R-Mesh. In section 5, we propose a constant time operation that enhances the generated path. Given an environment in the presence of obstacles, it is required to find a collision-free path from a source location *s*, to a destination location *d*. We assume that the obstacles to be disjoint convex or concave polygons. If two polygon images intersect, we consider them one polygon. The input to the R-Mesh is an $n \times n$ image that represents the environment. The image is digitized, stored to the R-MESH, one pixel per processor. There is a flag associated with each processor. The flag has a value of 1 or 0 based on the digitized image. We use an $n \times n$ R-Mesh to solve the path planning problem.

Generally speaking, the idea is to start with a straight line between the source *s* and the destination *d*. The final path tries to follow the line segment $\overline{sd}$. If $\overline{sd}$ does not intersect with any obstacle, then $\overline{sd}$ represents the final path. If $\overline{sd}$ intersects with a number of obstacles, then the final path follows the straight line segment $\overline{sd}$ with going around each

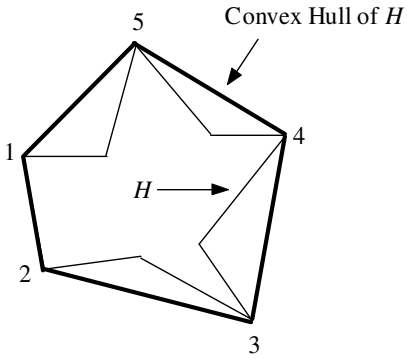


Fig.4 Enumerating the extreme points of a polygon *H*. The bold edges represent the convex hull of *H*.

obstacle that intersects with this path. We show that this path can be computed in constant time. Since going around an obstacle could be in clockwise or counter clockwise direction, then there are 2^m possible paths from *s* to *d* where *m* is the number of obstacles that intersects with segment $\overline{sd}$, $1 \le m \le k$.

Figure 5 shows the same environment as the presented in Figure 2. Figure 5 shows the path from the *s* to the destination *d* provided by the proposed algorithm.

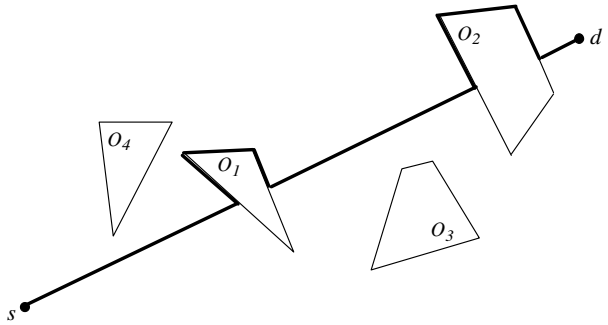


Fig.5 Path generated by proposed algorithm in bold

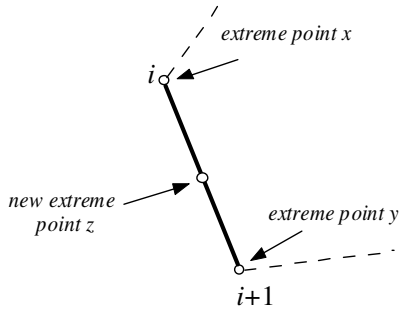


Fig.6 Addition of a new extreme point to a convex hull of an obstacle

4.1 Basic Operations

In this section we develop some operations on the R-Mesh that will be used in the proposed algorithm in Section 4.2.

- **Broadcasting Data**

Once the buses of R-Mesh are constructed, data movement on the R-Mesh requires constant time. The information written on the bus by any processor incident on the bus reaches all the processors incident on the bus in constant time. To broadcast data to all processors of the R-Mesh, partition all processors as $\overline{NSEW}$. If any processor write information on the any port, it reaches all processors in O(1) time. Thus broadcasting data to all processors requires constant time.

- **Adding a new point as an extreme point**

Let *H* be the convex hull of an obstacle *O*. Assume that the extreme points are enumerated in counter clockwise direction. Each extreme point knows its ID and the locations of preceding and following extreme points. In other words, each extreme point contains segments information associated with it. Consider an edge *e* with two extreme points *x* and *y* enumerated as *i* and *i*+1. The target is to add a new point *z* that is located on the segment as an extreme point. The operation uses the local bus via convex hull *H*. Let the processor representing *z* cut the local bus from *x* to *y*. The processor *z* exchanges the data with extreme points *x* and *y* using the local bus. The data exchanged include IDs of extreme points and their locations. Now processor *x* sets its following extreme point as *z* and processor *y* sets its preceding extreme point as *z*. Processor *z* sets its preceding and following extreme points as *x* and *y* respectively while the processors *x* and *y* keep their IDs. The processor *z* sets its ID to a special ID to indicate that *z* is an added extreme point. The described operations require constant time. Figure 6 shows the operation of adding a new extreme point to a polygon.

- **Constructing a bus from *s* to *d***

The target is to construct a bus from *s* to *d* on the R-Mesh that represents the segment $\overline{sd}$ as follows. Given that the locations *s* and *d* are known to all processors in the R-Mesh, each processor determines whether or not it belongs to the segment $\overline{sd}$. Then each processor that belongs to the segment $\overline{sd}$ computes the position of its preceding processor and the position of its following processor on the path from *s* and *d*. This allows each processor to configure its ports to connect its preceding processor and its following processor. For example, if a processor *p* finds that its preceding (resp. following) processor is the processor on its West (resp. North), then processor *p* configure its ports $\{\overline{NW}, E, S\}$ to connect the preceding and following processors. Constructing the bus includes a constant number of operations which can be done in constant time.

4.2 The Proposed Algorithm

In this section, we describe the proposed constant-time algorithm. We assume an $n \times n$ R-Mesh as the working environment. Again, the image of the obstacles is digitized, input and stored one pixel per processor. There are k disjoint polygonal obstacles. Also, we assume that the robot has been reduced to a point using the constant time algorithm proposed in [6]. Given the source point s and the destination point d, it is required to compute a collision-free path between s and d. Let P_s and P_d be the two processors that represent the positions of s and d respectively. The high level description of the proposed algorithm is as follows.

Collision-Free Path Algorithm

Input: The image of the obstacles, the positions of source s and destination d.
Output: A collision-free path between s and d.
Model: An N-processor, 2D-EREW R-Mesh

Step 1. Processors P_s and P_d broadcast their positions to all processors of the R-Mesh.
Step 2. All edges of all convex hulls calculate whether they intersect with the segment $\overline{sd}$ or not and determine the points of intersections if any. Let convex hull H_i intersects with $\overline{sd}$ at points l_i (l-point) and r_i (r-point) where $i \in \{1,..,k\}$. Let $S = \{H_i \mid i \in 1..k\}$ be the set of convex hulls that intersect with $\overline{sd}$ where $|S| = m$.
Step 3. For all $H_i \in S$, add points l_i *and* r_i to the extreme points of convex hull H_i.
Step 4. Construct a bus from s to d, *default bus*, and let all processors l_i and r_i cut the bus. Now, the bus from s to d consists of a number of segments. The first segment is from s to an l-point and the last segment is from r-point to d.
Step 5. Each two neighbouring processors cutting the default bus exchange their positions.
Step 6. Determine the collision-free path from s to d as a number of consecutive segments as follows.

- First segment is from s to the following l-point along $\overline{sd}$.
- The segment from any r-point processor to the following l-point processor is along $\overline{sd}$.
- The segment from any l-point processor to the following r-point processor is composed of a number of sub-segments along the convex hull of the obstacle and taken from the enumeration in the pre-processing.
- Last segment is from an r-point to d along $\overline{sd}$.

Now we describe the algorithm in detail. Step 1 broadcasts the positions of the source point and the destination point to all the processors on the R-Mesh. This enables all processors to compute the line segment $\overline{sd}$. In Step 2, each edge belongs to a polygon determine whether or not it intersects with segment $\overline{sd}$. If an edge e intersects with segment $\overline{sd}$, then the terminal points of the edge e determine the intersection point. Note that if a polygon H_i intersects with segment $\overline{sd}$, then it intersects in two points l_i (l-point) and r_i (r-point). The l-point (resp. r-point) is the one that is on the side of the source (resp. destination). Figure 7 shows four intersection points for the obstacles O_1 and O_2 with $\overline{sd}$. Let the set W contains all the intersection points; i.e. $W=\{l_i,,r_i \mid i \in 1..k\}$. Step 3 adds the two points l_i and r_i to the extreme points of the polygon H_i, $i \in 1..k$. Thus, each point l_i knows its preceding and following extreme point in the convex hull of the obstacle, same for point r_i. Since there are two paths around the obstacle to reach r_i from l_i, then a certain rotation direction should be decided. Here we assume, without loss of generality, that we go around the obstacle in clockwise direction. Note that the counter clockwise direction could be also followed. In Step 4, the algorithm constructs a bus, *default bus*, on the R-Mesh from s to d, as shown in Section 4.1 and let all processors l_i and r_i, $i \in 1..k$, cut the bus. In other words, the constructed bus now consists of a number of segments. There is a processor belongs to the set W between each two consecutive segments. In Step 5, each two neighbouring processors in W exchange their positions. This enables the start point, s, to know the following extreme point (an l-point) and also enables the last intersection point (an r-point) to know that the next point is the destination, d. Now the point s knows the first intersection extreme point (an l-point) and the last intersection extreme point (an r-point) knows the following point, d. Also along the path from s to d, each extreme point knows the following extreme point. Step 6 determines the final collision-free path from s to d. The path is composed of a number of segments. All the points l_i and r_i, $i \in 1..k$, belong to the final path and each represents a turning point on the path from s to d.

The first segment is from s to the following l-point and follows the segment $\overline{sd}$. The last segment is from an r-point to d and follows the segment $\overline{sd}$. A segment from any r-point processor to the following l-point processor follows the segment $\overline{sd}$. A segment from any l-point processor to the following r-point processor follows the convex hull of the obstacle and is taken from the enumeration in the pre-processing in Section 3.

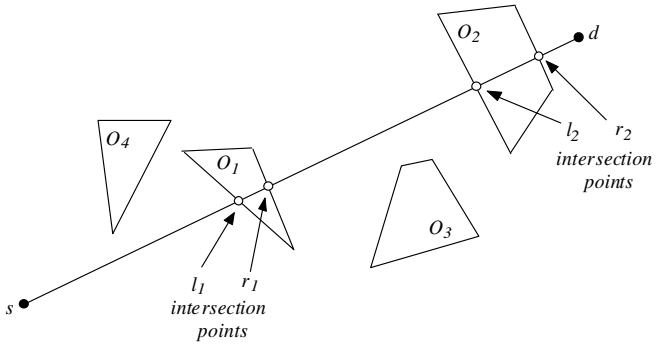


Fig.7 Intersection points for obstacles with segment $\overline{sd}$

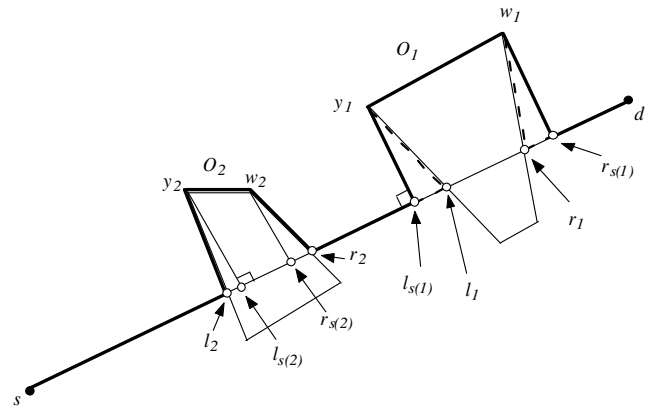


Fig.8 Replacing segments of the path generated by the proposed algorithm

4.3 Time Analysis

In this section, we show that the algorithm runs in constant time. Step 1 broadcasts the two positions for s and d to all processors of the R-Mesh. Broadcasting on the R-Mesh requires constant time as shown in Section 3. Step 2 calculates the intersection points for every edge with segment $\overline{sd}$. Since each extreme point in a convex hull contains the information of the two segments on which it is incident. Therefore, all the intersection points can be computed in parallel in O(1) time. Step 3 uses the sub bus within the polygon to add l_i and r_i to the extreme points of the polygon as shown before in O(1) time. In step 4, all processors use the positions of s and d to configure their ports in parallel such that the segment $\overline{sd}$ is constructed. The computation and configuration is done in constant time. Step 5 sends information on the constructed segments between each two neighbouring processors from Step 2. Sending constant number of variables on the bus requires O(1) time. Step 6 requires local computation at all points l_i and r_i, $1 \le i \le h$, $h \le k$ and this requires O(1) time. Thus, the whole algorithm runs in constant time.

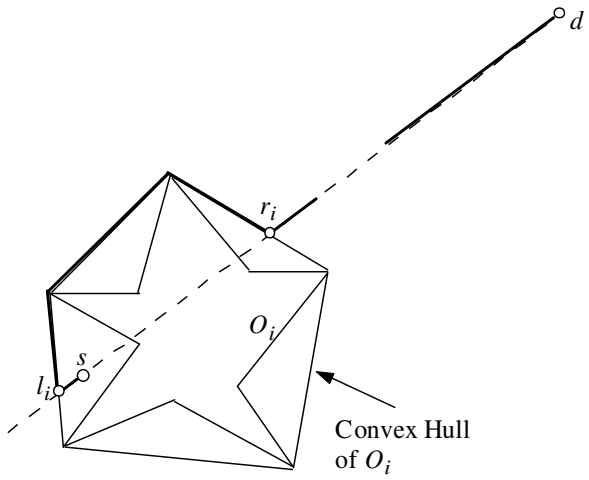


Fig.9 Handling the case if s or d is within a convex hull of an obstacle

5 Enhancing the path length

In this section, we present an enhancement to the proposed algorithm that could make a modification to the generated path. The target is to decrease the length of the generated path using constant time operations. The operations try to replace a segment of the path by another segment if this would shorten the path length. In particular, the operation tries to replace the segment $\overline{l_i y_i}$ of the path, by another segment $\overline{l_{s(i)} y_i}$ if the segment of $\overline{l_{s(i)} y_i}$ is outside obstacle O_i. Point $l_{s(i)}$ is the point of intersection between the segment $\overline{sd}$ and the perpendicular line to $\overline{sd}$ that passes through y. Figure 8 shows an example of the segment $\overline{sd}$ that intersects with two obstacles O_1 and O_2. The path generated by the algorithm is shown in bold. However, when applying the operation to the segment $\overline{l_1 y_1}$ of the path, this segment is replaced by another segment $\overline{l_{s(1)} y_1}$ that would make the path shorter. The same operation is applied to the segment $\overline{w_1 r_1}$ and is replaced by the segment $\overline{w_1 r_{s(1)}}$ because it happened to be of shorter length. If the new segment lies within the obstacle itself, then no replacement is done. This is the case for O_2 in Figure 8. The figure shows the path in bold when applying this operation to the path of Figure 5.

The above operation of replacing one segment by another can be done in constant time. Point y_i (respectively w_i) computes the point $l_{s(i)}$ (respectively $r_{s(i)}$) in constant time. If the computed point lies within the obstacle O_i then no replacement is done. If the computed point, for example $l_{s(i)}$, lies outside the obstacle O_i then the replacement would make the path shorter. In this case, y_i informs l_i and $l_{s(i)}$ by this replacement over the local buses in constant time.

6 Source and Destination within convex hulls

In section 4 we proposed the algorithm assuming that the source and destination points, *s* and *d*, are outside any convex hull. In other words, after preprocessing phase neither *s* nor *d* is located within any convex hull of an obstacle. In this section, we deal with the case that *s* or *d* or both are within a convex hull. Figure 9 shows an example where the source *s* is within convex hull of obstacle O_i. In such case, a minor change to the algorithm handles this situation. In Step 2 of the algorithm, after determining the intersection points, the source *s* and the first intersection point exchange information about their status. The status of a point could be an *l*-point (resp. *r*-point). If the source *s* found that the next point is an *l*-point, then the algorithm continues as described in Section 4. The *l*-point reaches the *r*-point through the convex hull of the obstacle. If the source *s* found that the next point is an *r*-point, then the source *s* decides that the next point on the path is on the other side from the destination point *d* and the algorithm continues. The above operation could also be applied for the destination *d* and it requires a constant number of steps and can be done in O(1) time. Figure 9 shows the generated path in bold in the case where *s* is within the convex hull of obstacle O_i.

7 Concluding Remarks

In this paper, we proposed an algorithm to compute a collision-free path between a source and a destination in an environment with the existence of obstacles. We used the EREW R-Mesh as the underlying architecture. The algorithm was shown to run in constant time and requires $O(\log^2 N)$ pre-processing time. This outperforms the previous method that required a linear time in the number of obstacles with the same pre-processing time. We also proposed a method that could shorten the length of the path generated by the proposed algorithm.

One possible extension to this work is to design algorithms that get the shortest path between the source position and the destination position in the existence of obstacles. Other directions include enhancing the preprocessing time and using other parallel models to solve the path planning problem.

References

[1] Y. Ben-Ashen, D. Peleg, R. Ramaswami, A. Schuster, The power of reconfiguration, J. Parallel Distr. Comput. 13 (2) (1991) 139–153.

[2] K.-L. Chung, H.-N. Chen, A neighbor-finding algorithm for bincode-based images on reconfigurable meshes, Comput. J. 43 (4) (2000) 315–324.

[3] F. Dehne, A. Hassenklover, J. Sack, Computing the configuration space for a Robot on a mesh-of-processors, in: Proc. 1989 ICPP, vol. 3, 1989, pp. 40–47.

[4] J. Jenq, W. Li, Computing the configuration space for a convex Robot on hypercube multiprocessors, in: Proc. 7th IEEE Symp. of Parallel and Distributed Processing, 1995, pp. 160–167.

[5] J. Jenq, D. Wang, Parallel computation of configuration space on reconfigurable mesh with faults, in: ICPP-2000 Workshop on High Performance Scientific and Engineering Computing, August 2000.

[6] J. Jenq, D. Wang, W. Li, Computing the configuration space on reconfiguration mesh multiprocessors, in: ISCA 13th Int. Conf. on Parallel and Distributed Computing Systems (PDCS-2000), August 2000.

[7] H.-C. Lee, Efficient parallel algorithms on reconfigurable mesh architectures, Ph.D. Dissertation, University of Missouri-Rolla, 1996. <http://www.mis.yzu.edu.tw/faculty/hlee/csdiss/>.

[8] R. Miller, V.K. Prasanna Kumar, D.I. Reisis, Q.F. Stout, Meshes with reconfigurable buses, in: Proc. MIT Conf. Advanced Research in VLSI, April 1988, pp.163–178.

[9] R. Miller, V.K. Prasanna Kumar, D. Reisis, Q.F. Stout, Parallel computations on reconfigurable meshes, IEEE Trans. Comput. 42 (6) (1993) 678–692.

[10] K. Nakano,``A Bibliography of Published Papers on Dynamically Reconfigurable Architectures," Parallel Proc. Letters, vol.~5, 1995, pp.~111--124.

[11] K. Sutner, W. Maass, Motion planning among time dependent obstacles, Acta Inform. 26 (1988) 93–122.

[12] P. Tzionas, A. Thanailakis, P. Tsalides, Collision-free path planning for a diamond-shaped robot using two dimensional cellular automata, IEEE Trans. Robot. Automat. 13 (2) (1997) 237–250.

[13] R. Vaidyanathan and J. L. Trahan, "Dynamic Reconfiguration: Architectures and Algorithms," Kluwer Academic/Plenum Publishers, January 2004.

[14] D. Wang, Two algorithms for a reachability problem in one-dimensional space, IEEE Trans. Syst., Man, Cybern. 28 (4) (1998).

[15] D. Wang, A linear-time algorithm for computing collision-free path on reconfigurable mesh, J. Parallel Comput. 34 (2008) 487–496.

The Mutual Exclusion Problem in Cellular Wireless Networks

Young-Whan Cho, Sung-Hoon Park, and Seoun-Hyung Lee
Dept. of Computer Engineering
Chungbuk National Univ., Chungbuk, Korea
{yhcho,spark}@cbnu.ac.kr

Abstract

The mutual exclusion (MX) paradigm can be used as a building block in many practical problems such as group communication, atomic commitment and replicated data management where the exclusive use of an object might be useful. The problem has been widely studied in the research community since one reason for this wide interest is that many distributed protocols need a mutual exclusion protocol. However, despite its usefulness, to our knowledge there is no work that has been devoted to this problem in a mobile computing environment. In this paper, we describe a solution to the mutual exclusion problem from mobile computing systems. This solution is based on the token-based mutual exclusion algorithm.

Key-words: Synchronous Distributed Systems, Mutual exclusion, Fault Tolerance, Mobile Computing System.

1. Introduction

The wide use of small portable computers and the advances in wireless networking technologies have made mobile computing today a reality. There are different types of wireless media: cellar (analog and digital phones), wireless LAN, and unused portions of FM radio or satellite services. A mobile host can interact with the three different types of wireless networks at different point of time. Mobile systems are more often subject to environmental adversities which can cause loss of messages or data [8]. In particular, a mobile host can fail or disconnect from the rest of the network. Designing fault-tolerant distributed applications in such an environment is a complex endeavor.

In recent years, several paradigms have been identified to simplify the design of fault-tolerant distributed applications in a conventional static system. Mutual exclusion, simply MX, is among the most noticeable, particularly since it is closely related to accessing shared resource called the critical section (CS) [7], which (among other uses) provides an exclusive access basis for implementing the critical section.

The mutual exclusion problem [1] requires two properties, safety and liveness, from a given set of processes. The problem has been widely studied in the research community [2,3,4,5,6] since one reason for this wide interest is that many distributed protocols need an mutual exclusion protocol. However, despite its usefulness, to our knowledge there is no work that has been devoted to this problem in a mobile computing environment.

The aim of this paper is to propose a solution to the mutual exclusion problem in a specific mobile computing environment. This solution is based on the token-based mutual exclusion algorithm that is a classical one for distributed systems. The rest of this paper is organized as follows: in Section 2, a solution to the mutual exclusion problem in a conventional synchronous system is presented. Section 3 describes the mobile system model we use. A protocol to solve the mutual exclusion problem in a mobile computing system is presented in Section 4. We conclude in Section 5.

2. Mutual Exclusion in a Static System

2.1 Model and Definitions

We consider a synchronous distributed system composed of a finite set of process $\Pi = \{p_1, p_2,., p_n\}$ connected by a logical ring. Communication is by message passing, synchronous and reliable. A process fails by simply stopping the execution

(*crashing*), and the failed process does not recover. A correct process is the one that does not crash. Synchrony means that there is a bound on communication delays or process relative speeds. Between any two processes there exist two unidirectional channels. Processes communicate by sending and receiving messages over these channels.

The mutual exclusion problem is specified as following two properties. One is for *safety* and the other is for *liveness*. The *safety* requirement asserts that any two processes connected the system should not have permission to use the critical section simultaneously. The *liveness* requirement asserts that every request for critical section is eventually granted. A mutual exclusion protocol is a protocol that generates runs that satisfy the mutual exclusion specification.

2.2 Token-based Mutual Exclusion Algorithm

As a classic paper, the token-based mutual exclusion algorithm, which was published by M. Raynal, specifies the mutual exclusion problem for synchronous distributed systems with crash failures and gives an elegant algorithm for the system; this algorithm is called the token-based MX Algorithm [2]. The basic idea in the token-based MX algorithm is that the any process holding the token can use the critical section exclusively. The token-based MX algorithm is described as follows.

- A distributed system is connected by a logical ring. Each process has a unique ID that is known by its neighborhood processes.
- The CS is exclusively used by the process holding the token.
- The token is circulated on the logical ring. If a process wants to use the CS, then it just waits until receiving a token from its neighborhood. Only when it has received the token, it has a right to use the CS exclusively.
- When the process with the token finished its use of CS, it immediately passes the token to its neighborhood.
- If a process doesn't to use a CS when it received the token, it just pass the token to it neighborhood.
- There exists only one token and the token is continuously circulated upon the logical ring.
- By doing this, any process eventually receives the token and it can use the CS exclusively, which means that this algorithm satisfies both of the safety and the liveness properties.

3. Mobile System Model

A distributed mobile system consists of two set of entities: a large number of mobile hosts (*MH*) and a set of fixed hosts, some of which act as mobile support stations (MSS_s) or base stations. The non *MSS* fixed hosts can be viewed as MSS_s whose cells are never visited by any mobile host. All fixed hosts and all communication paths connect them from the static network. Each *MSS* is able to communicate directly with mobile hosts located within its cell via a wireless medium. A cell is the geographical area covered by a *MSS*. A *MH* can directly communicate with a *MSS* (and vice versa) only if the *MH* is physically located within the cell serviced by the *MSS*. At any given instant of time, a *MH* can belong to one and only one cell. In order to send message to another *MH* that is not in the same cell, the source *MH* must contact its local *MSS* which forwards the messages to the local *MSS* of the target *MH* over the wireless network. The receiving *MSS*, in its turn, forwards the messages over the wireless network to the target *MH*. When a *MH* moves from one cell to another, a *Handoff procedure* is executed by the MSS_s of the two cells. Message propagation delay on the wired network is arbitrary but finite and channels between a *MSS* and each of its local mobile hosts ensure FIFO delivery of messages.

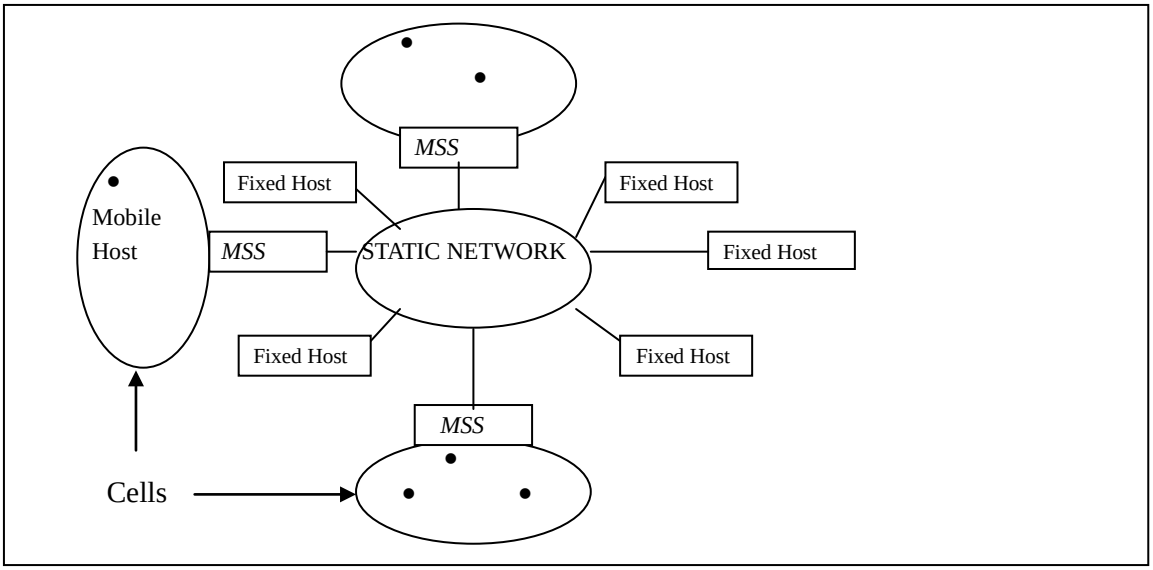


Figure 1: Mobile System Model

- The amount of computation performed by a mobile host should be kept low
- The communication overhead in the wireless medium should be minimal.
- Algorithm should be scalable with respect to the number of mobile hosts.
- Algorithm should be able to easily handle the effect of mobile host disconnections and connections.

4. Mutual exclusion in a Mobile System

In the following, we consider a broadcast group G = (G_MSS, G_MH) of communicating mobile hosts, where G_MH and G_MSS are respectively a set of m mobile hosts roaming in a geographical area (like a campus area) covered by a fixed set of n MSS_s. In so far, local mobile hosts of base station MSS_i, which currently residing in MSS_i cell, will refer to mobile hosts that belong to group G.

A mobile host can move from one cell to another. If its current base station fails, the connection between the mobile host and the rest of system is broken. To recover its connection, a mobile host must move into another cell covered by an operational or correct base station. So, unless it crashes, a mobile host can always reconnect to the network. A mobile host may fail or voluntarily disconnect from the system. When a mobile host fails, its volatile state is lost.

In this environment, the mutual exclusion problem is defined over the set G_MH of mobile hosts. When a mobile host h_k wants to use the *CS*, it sends the request message to a *MSS*. In this case, the mobile host eventually should get the permission from the *MSS* and use the *CS*. Due to the resources constraints of mobile hosts and the limited bandwidth of wireless links, the distributed algorithm to solve mutual exclusion is executed by the set of *MSS* on behalf of the set G_MH of mobile hosts. In a first phase, the *MH* which wants to use the CS has to request the permission from the *MSS* in the cell which it belonging to. The *MSS* receiving those requests from the subset of G_MH of mobile hosts roaming in their respective cells keeps them in its queue. A token is circulated through the logical ring which consists of the fixed *MSS*s. In the second phase, when each *MSS* receives the token from its neighborhood, it sends the token to a mobile host h_k to give permission for the CS.

Finally, the h_k received the permission from the *MSS* uses the CS and after using it returns the permission to the *MSS*. The *MSS* which has got the permission back from the h_k sends the token to the next turn of *MSS*s.

4.1 Principle

The mutual exclusion protocol proposed in this paper is based on the solution described by

Raynal in Token-based MX algorithm [2]. The outlines of their protocol have been described in Section 2. In this section, we give an overview of our protocol and identify the major differences compared with the original token-based MX algorithm. We assume that the mutual exclusion is initiated by a mobile host which requests its current base station a token to use the CS. The contacted base station saves the request into the queue until it receives the token from its neighborhood.

During the mutual exclusion, each base station on one hand interacts with the mobile hosts located in its cell to gather the request of each mobile host for CS and on the other hand interacts with the other neighboring base stations to send and receive a token. In our approach, a base station *MSS* which participates in the mutual exclusion protocol, always acts on behalf of a subset of mobile hosts.

More precisely, the initial value of $Token\_Holdek_k$ is false but the value of it is changed true as a mobile host h_k that resides in MSS_i receives the token from its MSS_i. After returning the token to its base station, the mobile host h_k changes the value of its $Token\_Holder_k$ into false again.

The mutual exclusion protocol in such an environment consists of two cases depending on who the token holder is. As the first case, that is when a base station received a token from its neighboring base station or its mobile hosts. When it received the token from its neighboring base station, then it just sends the token to a mobile host with highest priority among the mobile hosts connected to the base station. In case of returning the token from its mobile hosts, it just sends the token to the next base station.

In this case the base station is doing two tasks concurrently, i.e., sending a token a mobile host with highest priority among the mobile hosts connected to the base station and receiving the token request message from mobile hosts concurrently. Thus the base stations play a role of a mutual exclusion coordinator during the mutual exclusion period.

During the mutual exclusion in a mobile computing environment, a base station playing a role of a mutual exclusion coordinator is needed to reduce the message traffic among mobile hosts. The mutual exclusion algorithm among base stations is similar to the token-based MX in static distributed systems. That is, only the base station holding the token has a permission to use the *CS*.

In the second case, that is when a mobile host received a token from its host base station. Then it just uses the *CS* for a while and returns the token to its host base station after finishing it.

In above scenario, we don't consider the mobility of the mobile host in the MX algorithm. But if we consider the mobility of the mobile host, then it makes the MX problem more complicated than the one of static distributed systems.

The differences of mutual exclusions between mobile computing environments and static distributed systems are as follows:

1) During the period of the MH using the CS, the MH changes its base station from the one that it received the token to the other base station. In this case, the MH simply sends the token the base station of the cell in which it resides. But the base station that received the token takes some action to keep the fairness of the MX. The base station that did not send the token but received the token from its MH simply sends it to the base station which waits for the token to keep the fairness of the MX. Because, as a big difference between mobile computing environments and static distributed systems, the mobile host with token will appear in any cell whenever mutual exclusion protocol has started. Therefore, every base station should check all other base stations to know which base station cover the mobile host holding the token in the cell. That causes message traffics among base stations.
2) In mobile computing environment, a handoff algorithm is needed to perform mutual exclusions correctly, but it is not needed in static distributed systems.
3) Due to the resource constraints of mobile hosts and the limited bandwidth of wireless links, the distributed algorithm to solve mutual exclusion is executed by the set of MSS_s on behalf of the set *G_MH* of mobile hosts.

4.2 Protocol

The protocol is composed of three parts and each part contains a defined set of actions. Part A

(figure 2) describes the role of an arbitrary mobile host h_k. Part B (figure 3) presents the protocol executed by a base station MSS_i.

```
% Mobile host h_k is located in MSS_i cell %
(1) Upon receipt of the request for CS from
    the application
      Send Req_Token to MSS_i
(2) Upon receipt of Token from MSS_i
% The mobile host (h_k) gets into CS   %
      CS (h_k)
(3) Upon receipt of the release for CS from
    the application
      Send Release_Token to MSS_i
```

Figure 2: Protocol Executed by a Mobile Host h_k (Part A)

Part B is related to the interactions between a base station and its local mobile hosts on one hand and the other base station on the other hand. Thus, Part B is based on the traditional Token-based MX protocol adapted to our environment.

Finally, the part C of the protocol is the handoff protocol destined to handle mobility of hosts between different cells.

In figure 2, the three actions performed by an arbitrary mobile host are:

(1) A mobile host executes this action when it receives a request from an upper application program to initiate a mutual exclusion.
(2) Token message is sent to a mobile host h_k by the mobile support systems MSS_i when it had requested a token from the local base station where it resides. Upon receipt of such a message, the mobile host gets into the *Critical Section*.
(3) When the application program terminates the mutual exclusion protocol, the Token is released to the mobile support system, MSS_i.

Actions of the protocol in figure 3 numbered from (4) to (7) are executed a mobile support system, i.e., a base station MSS_i. They have the following meaning:

(4) When a base station is asked by a mobile host to send a Token, it inserts the request into the rear of its queue.

```
My_Staus_i := 0;
My_Queue_i := ∅;
Cobegin
(4) || Upon receipt of Req_Token( h_k )
       insert Req_Token(h_k) to rear (My_Queue_i);
(5) || Upon receipt of Token (MSS_i-1)
        if My_Queue_i ≠ ∅ then
          My_Status_i := 1;
           send Token to front (My_Queue_i);
           delete front (My_Queue_i);
        else
          send Token to MSS_i+1;
        end-if
(6) || Upon receipt of Token ( h_k )
      if ( Phase_i = 0 ∧ My_Queue_i ≠ ∅ ) then
           My_Status_i:= 1;
           send Token to front (My_Queue_i);
           delete front (My_Queue_i);
         else
           My_Status_i:= 0;
           send Token to MSS_i+1;
      end-if
(7) || Upon receipt of Req_Token ( MSS_j )
       insert Req_Token(h_k) to Rear(My_Queue_i);
```

Figure 3: Protocol Executed by a mobile support station MSS_i (Part B)

(5) In case of receiving a Token from other base station, the base station checks its queue *My_Queue*$_i$ to confirm whether the queue is empty or not. If the queue is not empty, then the base station sends the Token to the mobile host that is positioned at the front of the queue. And it deletes the element from the queue and sets its status to true that means it holding Token, i.e., *My_Status*$_i$:= 1. But if the queue is empty, then the base station just passes the Token to the next base station.
(6) When a base station receives a Token from a mobile host h_k, it checks its queue and status. If both (*Phase*$_i$ = 0 ∧ *My_Queue*$_i$ ≠ ∅) are true, which means that it does not hold the token and at the same time the queue is not empty, then the base station sends the Token to the mobile host that is the front element of the queue. And it deletes the element from the queue and sets its status to true. Otherwise it sends the Token to the next base station and sets its status to false.

(7) On receiving the Token request message from other mobile support system, the MSS_i insert the request message into its queue.

As shown in Figure 4, the handoff protocol is described.

(8) When a mobile host h_k moves from MSS_j cell to MSS_i cell, the handoff protocol execution is triggered. Mobile host h_k has to identify itself to its base station by sending a message GUEST(h_k, MSS_j).
(9) Upon receiving this message, MSS_i learns that a new mobile host h_k, coming from MSS_j cell has entered in its cell. With BEGIN_ HANDOFF(h_k, MSS_i) message, MSS_i informs MSS_j that it removes h_k from the set of mobile hosts that reside in its cell.
(10) Upon receiving such a message, MSS_j checks its queue to confirm that the token request of h_k is in the queue. If it is in its queue, then it transfers the token request to MSS_i and deletes the token request from the queue.

Cobegin
% Role of h_k %
(8) || **Upon** entry in MSS_i cell
s**end** Guest(h_k, MSS_j) to MSS_i
% Role of MSS_i
(9) || **Upon** receipt of GUEST(h_k, MSS_j)
Local_MH_i := Local_MH_i ∪ {h_k};
send BEGIN_HANDOFF(h_k, MSS_i) to MSS_j;
% Role of MSS_j
(10)||**Upon** receipt of BEGIN_HANFOFF(h_k, MSS_i)
$Local\_MH_j$:= $Loacl\_MH_j$ – {h_k};
If (Req_Token(h_k) ∈ My_Queue$_i$) **then**
send Req_Token(h_k) to MSS_i;
delete Req_Token(h_k) from My_Queue$_i$;
end-if

Figure 4: Handoff Procedure (Part C)

4.3 Correctness Proof

As our protocol is based on the Token-based logical ring algorithm proposed by M. Raynal, some statements of lemmas and theorems that follow are similar to the ones encountered in [2].

Theorem 1 No two different processes can have permission to use the critical section simultaneously (safety property).

Proof (proof by contradiction). Let assume that there exist two mobile hosts to get a permission to use the critical section. A mobile host can use the *CS* only if it received a permission token from the *MSS* of the cell to which it belonging (action 2). In this case, the assumption means that there exist two *MSS*s holding the token or one *MSS* sends the token twice to two different mobile hosts each. The first case is false since there is only one token circulating under the logical ring. The second case is also false since the *MSS* holding the token sends it to mobile host h_k only once (action 5). So it is a contradiction. □*Theorem* 1

Theorem 2 Every request for the critical section is eventually granted (liveliness property).

Proof If a mobile host sends a message to request a token (action 1), at least one *MSS* eventually receives it and inserts it into the queue (action 4). After that, there are two cases. In first case, if the mobile host h_k sent the message does not move to other cell, then the message Req_Token eventually will be positioned at the front of the queue and the *MSS* received the message sends the token. Thus, the mobile host sent the message eventually receives the token and uses the *CS*. In a second case, when the mobile host h_k sent a message Req_Token moves from MSS_j cell to another MSS_i cell before receiving the token, then the handoff protocol execution is triggered (action 8-10). Mobile host h_k has to identify itself to its base station by sending a message GUEST(h_k, MSS_j). In this case, by (action 10) the request message will be transferred to the MSS of the cell to which the mobile host has moved. Consequently, the mobile host will receive the Token and use the CS when the *MSS* sends the Token. □*Theorem* 2

5. Conclusion

The communication over wireless links are limited to a few messages (in the best case, three messages: one to request a token and the others to get the token and release the token respectively)

and the consumption of mobile hosts CPU time is low since the actual mutual exclusion is run by the base stations. The protocol is then more energy efficient. The protocol is also independent from the overall number of mobile hosts and all needed data structures are managed by the base stations. Therefore, the protocol is scalable and can not be affected by mobile host failures.

In addition, other interesting characteristics of the protocol are as follows. 1) During the mutual exclusion period, a base station should keep track of every mobile host within its cell to manage the request messages and the token. 2) In such a mobile computing environment, a handoff algorithm is needed to perform mutual exclusions efficiently and correctly, but it is not needed in static distributed systems.

The mutual exclusion algorithm in a mobile computing environment consists of two important phases. One is a local mutual exclusion phase in which a mobile host holds and uses the CS. The other phase is a global mutual exclusion phase in which each *MSS* takes part in the mutual exclusion by passing the token to another *MSS*.

References

[1] D. Agrawal, A.E. Abbadi, An efficient and fault-tolerant solution for distributed mutual exclusion, ACM Trans. Computing Systems 9 (1) (1991) 1.20.

[2] M. Raynal, Algorithms for Mutual Exclusion, MIT Press, Cambridge, MA, 1986.

[3] M. Maekawa, A √ *N* algorithm for mutual exclusion in decentralized systems, ACM Trans. Computer Systems 3 (2) (1985) 145.159.

[4] D. Manivannan, M. Singhal, An efficient fault-tolerant mutual exclusion algorithm for distributed systems, in: Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems, 1994, pp. 525.530.

[5] K. Vidyasankar, A simple group mutual *k*-exclusion algorithm, Inform. Processes. Letter 85 (2003) 79.85.

[6] M. Singhal, A taxonomy of distributed mutual exclusion, J. Parallel Distributed. Computing 18 (1) (1993) 94. 101.

[7] S. Lodha, A.D. Kshemkalyani, A fair distributed mutual exclusion algorithm, IEEE Trans. Parallel Distributed Systems 11 (6) (2000) 537.549.

[8] Pradhan D. K., Krichna P. and Vaidya N. H., Recoverable mobile environments: Design and tradeoff analysis. FTCS-26, June 1996.

[9] Alagar S., Venkatesan., Causally ordered massage delivery in mobile systems, in proc. Of Workshop on Mobile Computing Systems and Applications, Santacruz, CA, Dec. 1994.

[10] Badache N., Mobility in Distributed Systems, Technical Report #962, IRISA, Rennes, Octor 1995.

[11] Badrinath B.R, Acharya A. and Imielinski T., Impact of mobility on distributed computations, ACM Operating Review, 27(2), April 1993.

SESSION

GRID + CLOUD COMPUTING AND SUPPORTING TOOLS + APPLICATIONS

Chair(s)

TBA

Cloudlet Seeding: Spatial Deployment for High Performance Tactical Clouds

D. Shires, B. Henz, S. Park, and J. Clarke
U. S. Army Research Laboratory, Computational Sciences Division, APG, MD, USA.

Abstract— *Geospatially aware mobile devices, such as smart phones, rely on an architecture that is power constrained and processing power limited. The utility of these devices can be increased by offloading compute-intensive applications to parallel high performance computing (HPC) architectures, thus limiting battery drain, allowing access to large data, and providing faster time to solution. Such a paradigm can be achieved through tactical cloudlets that must operate in environments dominated by mobile* ad hoc *infrastructure (common in remote environments or military applications). Executing this paradigm is further complicated in that HPC nodes themselves (with some reduced mobility) are now deployable through the use of ruggedized hybrid core technologies. This paper discusses our concept for cloudlet seeding: the static strategic placement of HPC assets in deployed settings in such a way to balance computational load and limit hops to both stationary and mobile HPC nodes.*

Keywords: cloudlets, cloud computing, tactical computing, network emulation, high performance computing

1. Introduction

Army soldiers rely on mobile, fast, and up-to-date processing capabilities to be able to complete a myriad of mission assignments and tasks. This is often done in remote areas starved of computing and networking infrastructure or in urban environments where civilian infrastructure operates on differing protocols with security risks. Hence, mobile *ad hoc* networks (MANETs) will most often provide the operation frameworks for soldiers and military commanders [1]. It is within this operational realm that we are conducting basic and applied research to address how to field High Performance Computing (HPC) capacity within this MANET topology while at the same time increasing capabilities that leverage this new level of computing power. Coupling size, weight, and power with MANET-based operational requirements leads us to the establishment of tactical cloudlets as a way to deliver the required computing power to today's soldier [2].

Tactical cloudlets couple traditional connotations of cloud-based technology with the overarching goal of maximizing processing power and limiting the time to solution. This is all done in a mobile environment. Cloud computing has a history of focusing predominately on providing computing services with distributed processing and access to data. Commercial processing using high capacity network infrastructure paid little attention to time to solution as a key operational metric, although this is changing a bit with a new-found focus on streaming access speed to data. Military processing requirements push traditional cloud-based processing to new requirements in every dimension. For example, streaming content (such as sensor feeds) to a hand-held mobile smart phone poses numerous problems. First, forcing too much processing through mobile devices will quickly drain power. In a tactical cloudlet the option of piggybacking on other friendly devices may be beneficial. Second, the web of deployed devices must be flexible as signal loss and connectivity maintenance are known problems in deployed environments. Cyber foraging and possible process migration within the tactical cloudlet become critical. These issues are just the start of a long list of differences between commercial cloud systems and deployed tactical cloudlets.

The viability and overall success of tactical cloudlets depends largely on several intertwinded technical challenges including power-aware computing, cyber foraging (scheduling and process migration), and processing node optimization and characterization. In this paper we put forth another goal that should remain central in the development of tactical cloudlet technology; especially with regards to pushing HPC-level processing to the tactical edge. Tactical cloudlet seeding is an approach that recognizes the importance of spatial proximity for high performance servers to those mobile nodes that they serve. Simply put, cyber foraging will be greatly enhanced if assets are positioned in a way to limit hopping between and promote load balance within districts served by HPC nodes formed by this seeding approach. We discuss the importance of cloudlet seeding as a way to enhance the overall effectiveness of tactical cloudlets. In particular, we describe a methodology to perform cloudlet formation to enhance opportunities for cyber foraging algorithms within a realistic framework for mobile, ad hoc computing devices. This research directly impacts efforts to extend Army cloud computing and the capabilities of analysts and warfighters to the tactical edge.

2. Tactical Cloudlet Seeding

Unlike conventional developed infrastructure, usually consisting of a mix of wired and cellular networks, MANETS are often the only way of ensuring communication between entities deploying to remote areas or to areas with different infrastructure characteristics. Mobile networks are also important to the military mission as they can provide higher security profiles since the network is self-contained.

Deployed devices with the MANETS will be digital, vary in computational power, and operate over networks of varying data rates. Power figures heavily in this deployment as mobile nodes will be drained of battery at varying rates due to connectivity loss, network contention and collisions, computing load, and other operational conditions.

To address this problem, cloudlets have been proposed to address resource poor mobile hardware [3]. The central goal of this paradigm is to leverage computationally powerful resources with infinite power reserves by way of a cloud, but to do so by positioning the resources physically close to the mobile devices to limit the number of hops required for connectivity. That is, reduced latency between mobile device and data center will ensure longer battery life for mobile devices with fast access to information being processed by proximate computing resources.

Carrying this one step further, and by incorporating new heterogeneous and advanced computing methodologies, it is now possible to field near high performance computing (HPC)-level capabilities to soldiers and others in deployed settings. For military planners, HPC modeling and simulation have been useful tools for strategic planning. With tactical HPC using the cloudlet approach, dynamic and real-time processing of live data feeds can be possible for mounted and dismounted soldiers in operational environments. This can have real benefits to improve situational awareness and thus limit fog of war effects during mission execution.

This is a key point in our conceptualization of tactical cloudlet design. By incorporating HPC capacity, along with localized access to the fullest extent possible of mission-critical data at the HPC node, the overall bandwidth allocation dilemma faced by military planners if greatly mitigated. Bandwidth is a scarce commodity in deployed settings using MANETs [4]. Planning and optimizing the distribution is mainly relegated to a strictly strategic exercise subject to lack of complete information, possible exaggerated need from participants, and overall uncertainty. Cloudlet design with HPC-node centralization allows for a greedy-type optimization at the local level that can carry through to allow for an overall optimized network design.

However, access to these powerful devices will be key. While cyber foraging is a way to look for assets to help with processing demand, it does not address spatial locality of HPC to nodes requesting processing service. Tactical cloudlet seeding does just that; it attempts to localize access and balance demand on deployed HPC resources. Done properly, the overall goal is to limit foraging to predefined, seeded HPC districts where the processing load will be sufficient and balanced.

2.1 HPC Node Characteristics

Tactical networks of the future will be built upon a hierarchy of connectivity and capabilities. Hand-held radios and computing devices used by dismounted soldiers will obviously have different profiles and capabilities when compared to those housed in vehicles and stationary encampments. Overall connectivity will depend largely on signal strength, data transfer rates, contention, and positioning of nodes to reduce critical node formation.

Highly mobile compute nodes will always be lagging behind larger, less mobile nodes when it comes to sheer computing power. However, new hybrid architectures are being designed that allow for higher FLOP counts in shrinking footprints. These custom-engineered solutions can be ruggedized and provide a tiered capability for HPC in military applications. More fixed locations, such as Tactical Operations Centers (TOCs), have greater infrastructure and can support elaborate hybrid machines for applications tailored to them. Larger scale traditional multi-core systems, as well as more elaborate custom-engineered solutions like a seven card, water cooled, overclocked GPU-based system shown in Figure 1, can easily be supported in such a facility. Mobile and ruggedized GPU hardware can also easily be supported in platforms such as the High Mobility Multipurpose Wheeled Vehicles (HMMWV) or certain Unmanned Aerial Vehicles (UAVs).

Fig. 1: Customized GPU-accelerated workstation.

2.2 Network Goals

Tactical cloudlets have numerous measures of success measured largely on security and quality of service (QoS). Numerous techniques have been developed and proposed to enhance, improve, and ensure connectivity of MANETs [5], [6]. HPC nodes within the deployed MANET have special characteristics and specialized goals. While they should ideally posses access to large data rate channels, they should also be configured and distributed in a way that limits the number of hops that might be required of low-power hand-helds to reach them. In this regard, placement becomes critical as these resources should be spatially near assets that they serve. This is the main point of seeding the tactical cloudlets. Done properly, it will ensure speedy turnaround of queries and responses from powerful distributed computing nodes.

3. Problem Formulation

We are given a network based of mobile and fixed connected nodes. For the time being we assume that connections are static and once established are maintained during our period of interest. We are concerned with the connected graph $G = (V, E)$ that represents this network. In practice this will be a geographically constrained subgraph within a larger, distributed network. The vertices, V, represent the radios, computing devices, or other electronic systems capable of communicating within the system. These nodes will vary in computing capability, power requirements, etc. The edges, E, represent the communication links between these devices. We assume only one link edge between devices. The properties of these edges will vary based on data rates and other network characteristics. These parameters can become quite complex as various parts of this network will consist of mobile *ad hoc* nodes (MANETs) where geographic and weather effects can impact link capacity due to signal strength, packet loss, etc.

The vertex set, that is the networked components, is the set $V = 1, 2, \ldots, n$. We are concerned with fielding tactical HPC within the cloudlet infrastructure, hence we are actually attempting to determine how to classify the nodes of V. In this case, the set V is actually comprised of three disjoint subsets. These include the set of fixed HPC capable nodes, V_F, that could reside in deployed settings such as a Tactical Operations Center (TOC), the set V_H of mobile HPC nodes such as those deployable on vehicles, and the set V_M of mobile and/or deployed nodes that are not HPC capable. That is, we have V such that $\forall x \in V, x \in V_F \vee x \in V_H \vee x \in V_M$. For simplicity we assume nodes of overall equal properties within these subsets.

Strategically speaking, and for the static nature of the problem we are currently defining, the initial graph is constructed based on areas where HPC processing capability will be required, where HPC nodes exist in fixed locations, and other areas where dismounted soldier or other sensors and processors will be located. Node locations will therefore be a combination of known locations (such as the fixed nodes) and also areas where some type of coverage is required. Initial node locations will also be based historical or geographical data. In practice, HPC-capable mobile nodes will be positioned in safe areas, be restricted in total number due to cost or other issues, and be within a limited hop range from nodes requesting high rates of compute services. Any node that is not a fixed HPC node has the capability of being assigned to HPC mobile node status. Recognizing the fact that these nodes may reside in areas not amiable to that classification, we also define a set V_R that includes those node numbers that should not be allowed within the set V_H. Determining the initial graph configuration, including node placement and edge connectivity, is part of a larger research effort we are undertaking for tactical cloudlet configuration. How this is being performed through network simulation and emulation research will be discussed in a later section.

Our problem, therefore, is to assign every vertex v in V to an HPC node, whether it be mobile or fixed. All elements in V_M are assigned to an element in V_H or V_F. Since communication and data transfer happens along these graph edges, the path traversal becomes very important to fielding an ideal system. The weight of any path $p = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_m$ is the sum of the weights of the edges that it contains:

$$w(p) = \sum_{i=1}^{m} w(v_{i-1}, v_i) \tag{1}$$

The shortest path from u to v is therefore defined as:

$$\delta(u, v) = \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} \tag{2}$$

3.1 Constraints

Since we only have a limited number of HPC-deployed nodes, our goal is to assign every node in our graph to some HPC-computing node. That is, we are attempting to determine a partitioning of the set V into k disjoint subsets $V_1, V_2, \ldots, V_k$ to form a k-way partitioning of V. In this case k will be equal to $|V_F| + |V_H|$.

Optimal deployment of HPC computing capability to the tactical edge will require the solution of a multi-constrained problem. First, we deal with path traversal or the number of hops a device must travel to reach HPC capable computing nodes. Ideally this number should be as small as possible for several reasons, including amount of data that must be transferred, risks of intermediate node drop-off, etc. Let u_i be the HPC node within partition i. We are searching for a total minimum hop count across all k partitions:

$$\min \sum_{i=1}^{k} \sum_{j=1}^{|V_i|} w(u_i, v_j) \tag{3}$$

In certain cases, it may also be necessary to balance the data rates as well as hop counts. Depending on the amount of data stored local on mobile hand-helds versus the HPC node, the impact of these network links will vary considerably. Scalar data based on GPS location, or the transmission of static browser images, will require less total bit transmission with delay and hop counts taking higher priority. We see this as the more common operational case, but are currently in the process of further modifications to equation 3 to better account for those cases where data rates retain a high importance. These cases pose extra difficulties since overall data movement capabilities between two nodes is restricted to the bottleneck link in the path. In the current formulation we only concern ourselves with hop counts or distance from HPC node to the nodes it serves. That is, $\forall e \in E, w(e) = 1$.

Second, we want the overall offered computing load to be balanced within the k-way partitions. Doing so will ensure that HPC resources will not be overwhelmed by the nodes that they serve. This computational load is analogous to the partition weight of the ith partition, denoted by $w(V_i)$, which is equal to the sum of the weights of the vertices in that partition. Load imbalance then becomes a critical metric and a number that needs to be minimized. In a k-way partition, this load imbalance, LI, is a ratio of the highest partition weight divided by the average partition weight [7].

With some generality and assuming an average, distributed computational load, we arrive at $\forall v \in V, w(v) = 1$. An ideal distribution of computing load is hence analogous to seeking approximately equal cardinality of the subsets formed by the k-way partitions.

4. Tactical Cloudlet Seeding

Using the constraints developed in the previous section, we describe the approach specified in algorithm CLOUDLET-SEEDING that can be used to specify the location of HPC mobile nodes within the deployed network. The inputs to the algorithm include the $n \times n$ adjacency matrix A of the graph G, the set V_F of fixed HPC nodes, the set V_R of restricted nodes that cannot be designated as mobile HPC nodes, and the scalar q representing the total number of mobile HPC nodes that are allowed.

The algorithm builds the matrix $D[1\,..\,n, 1\,..\,n]$ that gives the shortest path distance between all pairs of vertices. The algorithm also builds and iterates over the power set (set of sets) $B = \{x \in 2^V : (|x| = q) \wedge (\forall y \in x, y \notin V_F \wedge y \notin V_R)\}$. The matrix B is therefore all possible q-way combinations of the vertices in V excluding the nodes that cannot serve as HPC mobile nodes and those fixed HPC nodes. The number of these sets is given by the binomial coefficient and in this case is $\binom{n-(|V_F|+|V_R|)}{q}$.

In order to hold information about the various partitions, we also define a partition type p_type containing a scalar value of the HPC node in that partition, a scalar keeping count of the number of nodes assigned to that partition, and a list of those nodes assigned to the partition. The variables p and Z are of this type.

The procedure makes use of several ancillary functions. The function INIT-PART returns an initialized array of structures of length $q + |V_F|$ containing information as defined in the p_type data structure. At each iteration of the loop, this structure is initialized with the fixed HPC node location data and also the current combination of possible HPC nodes selected from the set B. MIN-HOPS is a straightforward iteration down the rows of shortest path matrix where every possible HPC vertex location specified in p is compared against the other for hop counts. In cases where there is a clear winner, the vertex is assigned to that partition immediately. The case where the row happens to correspond to one of the possible HPC locations is a clear example. In this case the hop value will be 0 and that vertex is assigned to that corresponding partition. In cases where there are more than one possible HPC locations with the same minimum hop, the decision is delayed until the function BAL-PART. At this time, the value of $part.count$ becomes important. Partitions with a lower total weight (number of vertices served) win and are assigned that row. In cases of further ties, we arbitrarily pick the lowest numbered vertex as the winner. The procedure returns a partition structure Z that tells the q-way partition assignment of $a[v_1\,..\,v_n]$.

```
CLOUDLET-SEEDING(G, V_F, V_R, q)
 1  A ← ADJACENCY-MATRIX(G)
 2  D ← ALL-PAIRS-SHORTEST-PATHS(A)
 3  h ← ∞
 4  for i ← 1 to (n−(|V_F|+|V_R|) choose q)
 5       do p ← INIT-PART(i, V, V_F, V_R, q)
 6          for j ← 1 to n
 7               do MIN-HOPS(p, D)
 8          for j ← 1 to n
 9               do BAL-PART(p, D)
10          if HOPS(p) < h
11            then h ← HOPS(p)
12                 Z ← p
13            else if (HOPS(p) = h)
                   ∧(LI < LOAD-BAL(p))
14                   then h ← HOPS(p)
15                        Z ← p
16          return Z
```

Consider the graph as shown in Figure 2 as an input to this algorithm. We assume uniform computational load on all nodes and equal weight on all edges. Nodes 4 and 7 are special; they represent fixed HPC-enabled compute nodes within this graph. All other nodes in the graph represent locations for possible mobile HPC placement (the vector r is empty). In this case we can field one mobile HPC node ($|V_H| = 1$). We start by computing the all-pairs shortest paths to determine hop distance between the nodes. Since HPC nodes under consideration or those fixed will have a hop distance of 0, they will by default be the HPC element of their respective partition. We then generate the possible subsets of size 1 (nodes 1, 2, 3, 5, 6, 8, 9, 10, 11, 12, and 13). Each of these is analyzed according to the criteria in lines 7 and 9. On the first pass, nodes with a clear shortest hop to an HPC node are assigned to that node's partition. On cases of a tie, the assignment is deferred to the next pass. Here, the weight of both partitions is assessed and the node is assigned to the partition with the lower weight. The weight of this partition is then incremented by that node's weight.

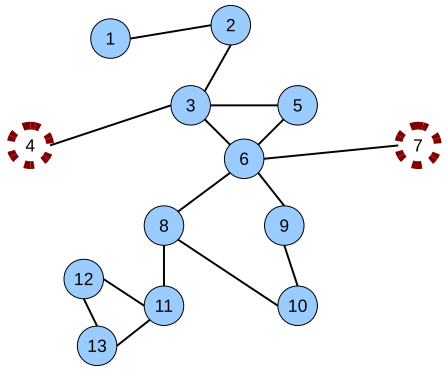


Fig. 2: Example MANET connectivity graph. Nodes 4 and 7 are HPC fixed locations.

The result of this algorithm is shown in Figure 3. Partition 1 is based at HPC node 4, partition 2 at HPC node 7, and partition 3 at HPC node 11. The solution gives a path no longer than 3 in partition 1 (P1, total hops = 6), no longer than 2 in partition 2 (P2, total hops = 5), and no longer than 2 in partition 3 (P3, total hops = 5). The total hop count is 16 for this partitioning. Partition weights are 4, 4, and 5 for partitions 1, 2, and 3 respectively. The load imbalance in this 3-way partition is 15%.

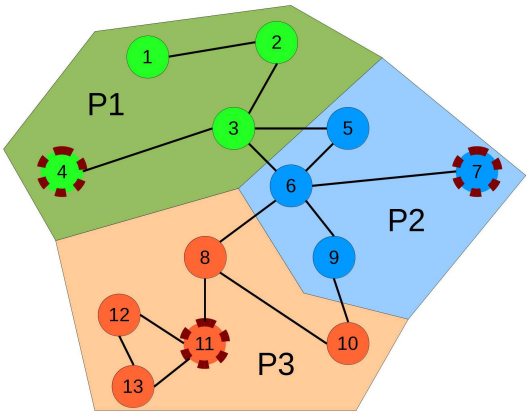


Fig. 3: An optimal three-way partition of the graph in Figure 2. Nodes 4 and 7 are assisted by a mobile HPC node at location 11.

Contrast this to the worst mobile HPC node assignment discovered by the algorithm which is shown in Figure 4. The mobile HPC node in this case is node 5. LI is also 15% (not uncommon for such a small problem set), the total hops is 25 (4 for partition P1 serviced by node 4, 10 for partition P2 serviced by node 7, and 11 for partition P3 serviced by node 5). The maximum path length in this case is 4. There is another major problem with this solution. The cloudlet concept of localized access to HPC nodes is severely violated. Several nodes, such as node 1, require connectivity through external cloudlets to connect to their assigned HPC resource. Such a situation will probably be experienced, at least transiently, in a tactical situation. However, strategic solutions should not produce such assignments and routing.

While nodes v_1 and v_m within the path $p = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_m$ will be members of the same partition set, more rigorous analysis will be needed to verify that nodes $v_2 \ldots v_m$ will reside in the same partition. In a static analysis, this is not overly important as membership to a partition is not as important as minimizing the overall hops and load imbalance. It may become more important if further refinements for dynamic optimization require members of partitions to move in such a way to maintain connectivity.

4.1 Analysis

The algorithm CLOUDLET-SEEDING is a combination of all-pairs shortest path, brute-force methods for generating the possible subset configurations, and a 2-phase peephole greedy method optimization for resolving partition assignment. The data structures required will generally include

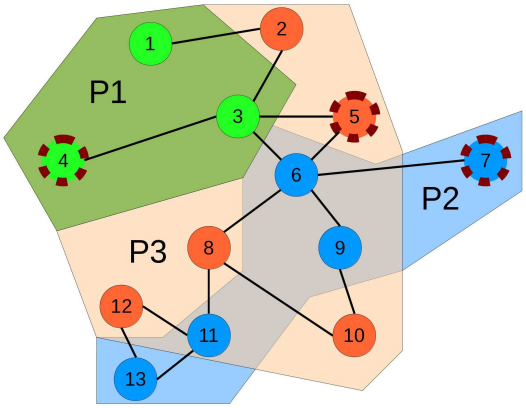


Fig. 4: A poor three-way partition of the graph in Figure 2. Nodes 4 and 7 are assisted by a mobile HPC node at location 5.

simple scalars, vectors of size n, and arrays and structured data types not exceeding sizes of $n \times n$. Space can also be conserved through the use of more elaborate sparse structures such as linked lists.

We can compute a simple upper bound on the running time of CLOUDLET-SEEDING. Forming the adjacency matrix takes $O(n^2)$ since there can be at most one edge between any two vertices. Ideal computation of all pairs shortest path will depend on restrictions to edge weights (currently uniform at cost 1). Dijkstra's algorithm or the Bellman-Ford algorithm will suffice with varying data structures impacting overall performance (binary heap versus Fibonacci heap in the binary-heap implementations). In general though this runtime will be $O(n^2 \lg n)$. These times are all incurred once outside of the main loop.

The generation of the combinations of subsets of size $q-(|V_F|+|V_R|)$ becomes more complex and corresponds to the beginning of the main processing loop (iteration counter i). The function INIT-PART is responsible for generating and initializing partitions for evaluation as HPC nodes. This function will omit nodes already selected as stationary HPC nodes (V_F) and nodes that are restricted (V_R). This function could generate these subsets by maintaining a list of static pointers to the vertices where these pointers are incremented at each call for the next iteration's subsets. Of primary concern, however, is the iteration count to generate these combinations. In practice we expect the number of HPC nodes to be a very small percentage of the overall number of fielded binary systems, and we assume the number of fixed HPC nodes and restricted nodes are relatively small. Since the number of HPC nodes within the problem set will be small, we recognize that q will be some fraction of n. We can then rewrite $\binom{n}{q}$ as $\binom{n}{\lambda n}$, where $0 \leq \lambda \leq 1$, with a bounds of $O(2^{n\,H(\lambda)})$ where H is the binary entropy function $H(\lambda) = -\lambda \lg \lambda - (1-\lambda) \lg (1-\lambda)$ [8].

The calls to procedures MIN-HOPS and BAL-PART each must iterate over all rows of the shortest path matrix doing simple minimization comparisons and assignments. Thus

the time for each of these is $O(n\,(q + |V_F|))$. HOPS and LOAD-BAL simply return the current hop count and nodal balance of the partitions, respectively. The overall runtime of CLOUDLET-SEEDING is thus $O(n\,(q + |V_F|)2^{n\,H(\lambda)})$.

Some trivial optimizations are possible. For instance, should the result of MIN-HOPS yield a result greater than the already established minimum hop count, there is no need to try to balance the partitions. Rather the next iteration of the outer loop can begin immediately.

5. Initial Graph Construction and Optimization

The current static formulation for cloudlet seeding requires some initial configuration of network asset placement with a corresponding assessment as to how the nodes will be linked. The heterogeneous nature of the deployed binary computing devices, from unattended sensors to powerful systems in areas with supporting infrastructure, makes this assessment both tedious and difficult. Often this is done by hand and through empirical data. Locations are based on historical, geographical, meteorological, and sometimes "best guess" estimates based on characteristics of the devices to be deployed.

We are currently designing a system based on network emulation with real-time radio frequency propagation analysis to determine signal loss of devices communicating wirelessly and determine the impact on data transmission to HPC nodes. This will require modifications to the cloudlet seeding approach where communication edges could be weighted to better account for the physical problem.

Furthermore, we are also performing calculations based on line-of-sight (LOS) as a threat assessment tool in an effort to determine areas that are out of harms way. These areas will in most cases be the better candidates for districting HPC resources. Furthermore, with a feedback mechanism built into the network emulator, areas will be chosen such that they provide shelter and enhanced network connectivity. Tactical cloudlet seeding therefore quickly grows into a multifaceted constraint-based optimization problem.

6. Future Work

While this initial assessment attempts to keep the cardinality of the subsets formed roughly equal and limit hops, we recognize that further research is justified as we move forward in our goal to push HPC to the edge. For example, we may wish to specify numerous characteristics to the edges of this connectivity graph. Bandwidth and latency characteristics will differ significantly based on the properties of the nodes (radios versus sensors versus etc.). It should also be noted that the computational capabilities of the nodes in the system will be highly disparate, and the needs of different resources will need to be further developed and represented to the optimization engine. Computational request load will create a dynamic situation where the weights of the nodes will need to be considered. Computing load and demand must also be further defined.

Furthermore, the current strategy serves a strategic goal and is static in nature. We are looking into ways to extend methods to tactical scenarios that are dynamic. Changes on the ground may require mobile HPC assets to be moved. Failure of certain nodes, due to loss of battery or signal loss due to movement, will necessitate a need to recompute HPC-directed network traffic. Compensating for these situations will greatly extend the capabilities of this system.

The computational complexity for optimization problems such as this highlights the need for heuristics especially moving into dynamic real-time adjustments to the deployed network. Accordingly, we plan on investigating other approaches such as those found in multi-level partitioning in graph partitioners to reduce the time complexity of CYBER-DISTRICT [9]. Discovering ways to include parameters such as shortest path distance, along with other parameters deemed important from network science research, into these heuristic-based methods will greatly enhance confidence in an optimally deployed system.

7. Conclusions

Situational awareness and applications demanding at or near real-time processing speeds will continue to push the capabilities of hand-held computing devices. This will continue to be the case even as processor technologies for these devices continue to improve. However, with advanced computing architectures constructed of hybrid accelerators, a new higher level of computing is possible and can be deployed in operational and mobile environments. Accessing these technologies, connected via wired and wireless networks, becomes key as hand-held devices try to off-load processing to both save power and increase computational efficiency.

Tactical cloudlet seeding puts forth an important augmentation to concepts in cyber foraging. Seeking to reduce network node hops in reaching an HPC-enhanced node and balance the workload of these servers, it provides a methodology to greatly increase the abilities of mobile resources in resource-poor areas. We have laid the initial groundwork for this concept, and look forward to expanding the challenging optimization problems that it presents for both static and dynamic, strategic and tactical total network capability.

References

[1] M. D. Kristensen, "Execution plans for cyber foraging," in *Proceedings of the 1st workshop on Mobile middleware: embracing the personal communication device*, ser. MobMid '08. New York, NY, USA: ACM, 2008, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/1462689.1462692

[2] G. F. Carey, E. Barragy, R. McIay, and M. Sharma, "Element-by-element vector and parallel computations," *Comm. Appl. Num. Methods*, no. 4, pp. 299–308, 1988.

[3] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE*, vol. 8, no. 4, pp. 14 –23, oct.-dec. 2009.

[4] X. Su, S. Chan, and J. H. Manton, "Bandwidth allocation in wireless ad hoc networks: challenges and prospects," *Comm. Mag.*, vol. 48, no. 1, pp. 80–85, Jan. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1820984.1820999

[5] Z. Han, A. Swindlehurst, and K. Liu, "Smart deployment/movement of unmanned air vehicle to improve connectivity in manet," in *Wireless Communications and Networking Conference, 2006. WCNC 2006. IEEE*, vol. 1, april 2006, pp. 252 –257.

[6] R. Hunjet, A. Coyle, and M. Sorell, "Enhancing mobile adhoc networks through node placement and topology control," in *Wireless Communication Systems (ISWCS), 2010 7th International Symposium on*, sept. 2010, pp. 536 –540.

[7] A. Abou-rjeili and G. Karypis, "Multi-level algorithms for partitioning power-law graphs," University of Minnesota, Tech. Rep. TR 05-034, 2005.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, 1st ed. The MIT Press, 1990.

[9] G. Karypis, "Multi-constraint mesh partitioning for contact/impact computations," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 56–. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050206

A Cloud System Implementation for the Analysis of Civil Engineering Structures

J.M. Alonso[1], A. Alonso[2], P. de la Fuente[1], F. Gómez[2], V. Hernández[1], P. Lozano[1], A. Pérez[2]
[1]Institute of Instrumentation for Molecular Imaging, Universitat Politècnica de València, Valencia, Spain
[2] Department of Mechanics of the Continuous and Structure Theory, Universitat Politècnica de València, Valencia, Spain

Abstract - *This paper describes the development of a high throughput and reliable Cloud Service to perform on-demand structural analysis over the Windows Azure-based Cloud infrastructure provided by the EC VENUS-C project. All the simulations in the Cloud are governed by Architrave®, an advanced software environment for the design and analysis of buildings and civil engineering structures. The migration to the Cloud has been implemented by means of the Generic Worker component, a web-role implementation for Windows Azure that manages the execution of the remote tasks. The CDMI standard has been used for uploading and downloading data. A GUI Client has also been implemented, in charge of defining and managing the remote simulations, transferring the data and informing the user about the status of the simulations. A real large building has been chosen as a case study to show the advantages of the cost-effective Cloud system with respect to the sequential approach.*

Keywords: Cloud Computing, Cloud Service, structural analysis, VENUS-C Project

1 Introduction

Structural analysis of buildings, or civil engineering structures, is the process to determine the response of a structure to different prescribed applied loads. This response is usually measured by establishing the stresses, tensions and displacements at any point of the structural elements.

In a linear dynamic analysis [1], where the external loads (earthquake, wind load, etc.) change along the time, the second order differential equations in time that governs the motion of structural problems must be solved. Direct time integration algorithms are techniques usually applied for solving this computationally demanding equation of motion, using a time step-by-step numerical integration procedure that provides the response of the structure along the time. The accuracy of the results depends on the time increment employed.

The realistic 3D structural dynamic analysis of large scale structures can demand an important computational power, give place a huge volume of data and become one of the most time consuming phases in the design cycle of a building or a civil engineering structure. For this reason, this analysis has been traditionally solved by introducing a variety of simplifications (unsuitable for complex structures) in order to reduce the problem size and the volume of the data, and obtain the results in reasonable simulation times.

Architects and structural engineers need thus powerful software applications able to simulate efficiently the accurate response of the structure. High Performance Computing (HPC) techniques provide powerful numerical and programming tools to develop applications able to simulate, efficiently and in a realistic way, large dimension structures, in very reasonable response times. However, commercial applications usually offer traditional approaches, computing sequential structural analysis on the user's local machine. As a result, the size and the complexity of the structure to be analysed, the type of structural analysis employed and the total number of the different structural solutions or even earthquakes evaluated are limited by the performance of the computational resources used by the users. With the purpose of overcoming these limitations, Architrave® [2], an advanced software environment for the design, 3D linear static and dynamic analysis and visualisation of buildings and civil engineering structures, was developed. Architrave is composed of these three different and independent components, although interacting among them:

- The Design Component: An interactive AutoCAD-based application where the user can draw the model and define the structural properties and the external loads.
- The Analysis Component: An interactive GUI application, where the user can modify the structural properties and the external loads, analyse the structure and visualise the results.
- The Structural Simulator: A batch MPI-based parallel application used by the Analysis Component to simulate the response of the structure by means of the Finite Element Method.

Notwithstanding, studios for architecture and engineering rarely own parallel platforms to execute an HPC-based application. Thus, the users install and run Architrave in their personal computers, and the time spent on the calculations by means of the Structural Simulator component depends on the performance of their machines.

Fortunately, Cloud Computing technology has emerged as a solution for the computational requirements of organizations, enabling the usage of non-owned remote resources which delivers enough power and storage space to satisfy the computational and disk requirements of the resource-starved dynamic structural simulations of high-rise buildings. Moreover, Cloud technology allows sharing not only computing power, but also another kind of resources such as storage space, data and even software packages.

This document is composed of the following sections. First of all, the VENUS-C project and the Generic Worker component are described in Section 2. The porting of a structural analysis application to the Cloud is explained in Section 3. This section describes the design adopted for the Generic Worker execution model and the software architecture. The different VENUS-C components used are explained and the details of the deployment in the platform tested are finally exposed. Section 4 presents the representative case study selected in order to validate the solution implemented in this work. Finally, the Section 5 contains the conclusions.

2 The VENUS-C project and the Generic Worker component

VENUS-C (Virtual multidisciplinary EnviroNments USing Cloud infrastructures) [3] is a European Research Infrastructure Project that aims at providing an easy-to-use computing platform, based on the virtualisation and service orientation. In simple terms, it allows researchers to use the Cloud Computing model in science.

VENUS-C is associated with the concept of PaaS (Platform as a Service), which means that an application is inserted into an existing Virtual Machine (VM) image with an appropriate runtime environment, and then executed. The developer need not to maintain, manage or update the VM's operating system or runtime environment. Instead, he focuses on the functionality of his application. Also, this application can be easily scalable, secure and ensure high availability.

This is what provides other platforms like Windows Azure [4] or Google App Engine. However, the novelty of this initiative is, among other ones, the easiness of porting an existing application to the Cloud and the interoperability between different infrastructures without having to adapt the application. In this regard, VENUS-C exposes an OGF BES/JSDL [5][6] compliant web service interface and client libraries for Java and .NET.

VENUS-C is a platform composed by several components which provide different services. The two essential services are data and job management. The data management service includes transferring input data to the Cloud and retrieval of output results. It supports the Cloud Data Management Interface (CDMI) specification [7], developed by the SNIA, and considers also blobs from Windows Azure Storage for transferring data.

The job management service allows the scientist to allocate compute resources in the Cloud, submit tasks, and to manage a job's lifecycle, i.e. to monitor the execution status and terminate a job if necessary. Other services provided are Elasticity, Monitoring, Accounting, Billing, and traffic redundancy elimination.

There are two components available that support the job management service (Figure 1): the Generic Worker (GW) [8][9] from Microsoft Research, and the PMES-COMPSs [10] from the Barcelona Supercomputing Center (BSC). Each of these components has a different programming model. In the implementation for this work, the first of them has been used, which is intended to be deployed on the Windows Azure platform, running over VMs with Windows Server 2008 and .NET framework.

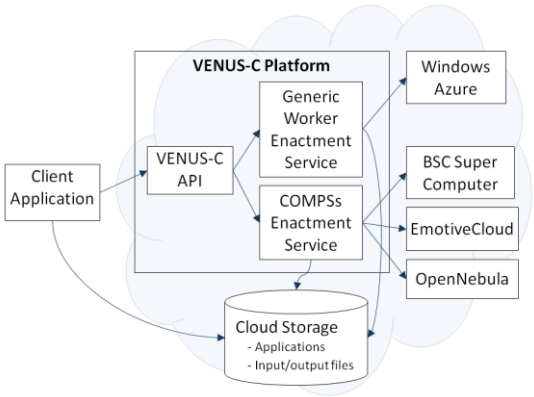


Fig. 1. The basic architecture of VENUS-C.

Actually, the GW is a Windows Azure web role attending for requests of registering a job, getting the status of a task, terminating a job, etc. (supporting, in this way, the VENUS-C API). In addition, this same web role has another process running that checks for new jobs registered to be processed. Obviously, one or many instances of this web role can be deployed working together in coordination.

The GW provides the common glue code that the developers have to write in order to port their applications to Windows Azure. The difference is that this service executes *generic* jobs, i.e. every job has a description that specifies the application to run, the input/output files and the parameters to be passed to the executable binary.

All interactions with the GW can be authenticated by the username and password mechanism and controlled by authorization policies, based on certain user roles previously defined and a table that specifies which roles has each of the users. Also, the communications can be protected through WS-Security with security tokens.

The GW exposes an additional interface to the administrator, through which the number of running instances can be scaled up/down to ensure the computational resource demands of the clients. Thus, the scaling decisions can be taken automatically according to different criteria.

3 Design and implementation

This section describes the design adopted to offer the capability to carry out remote structural analysis in the Cloud. The implementation was structured in different components and layers considering usability and easiness to adaptation to the features provided by the VENUS-C software.

The architecture of the Cloud Service implemented, based on the GW component, is shown in the Figure 2.

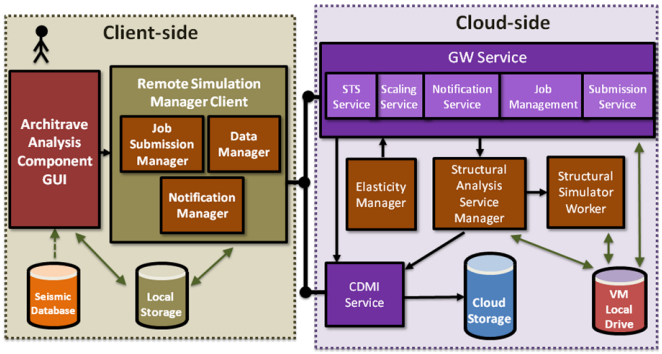


Fig. 2. Software architecture of the GW-based Cloud system.

Firstly, it consists on a Windows client that executes the Architrave Analysis Component to modify the structural properties, apply the external actions, define the simulation and visualise the results. In addition, a GUI tool, called as the Remote Simulation Manager Client, has been implemented to submit and manage the simulations in the Cloud and receive the results. Data communications between the local client and the Cloud service are carried out by means of the standard CDMI service [11]. On the Cloud-side, user authentication is implemented by means of the Security Token Service of the GW, and a Notification Service, also belonging to the GW, is used to inform the user about the status changes of the simulations. Remote jobs are managed and submitted thanks to the Job Management and Submission Service components of the GW. Structural simulations in the Cloud are executed by means of the Architrave Structural Simulator component. A Structural Analysis Service Manager module, in charge of launching the Architrave Structural Simulator Worker with the appropriate parameters, depending on the type of analysis, and managing the needed simulation input and output data has been implemented. Finally, an Elasticity Manager component has been developed to provide the system with the elasticity capability, using the Scaling Service of the GW. A more detail explanation is given in the following subsections.

3.1 The Remote Simulation Manager Client

On the client-side, the local or remote simulations are defined exactly in the same way, using the Architrave Analysis Component, the GUI application where the structural properties can be modified and response of the structures is visualized, after the simulation. This application now incorporates the Remote Simulation Manager Client, a tool developed also for .NET with a comprehensive GUI to configure and manage the simulations in the Cloud, to know in real-time the status of the remote executions and to download the results. This new application is launched by Architrave Analysis Component as an independent executable binary, which can even continue running (for downloading results, for example) when Architrave Analysis Component application is closed. In this tool, some modules can be distinguished:

- The Job Submission Manager: When the Remote Simulation Manager Client is launched, for the first time, this module checks the status of all the simulations in the Cloud (submitting, pending, running, finished, failed, downloading, downloaded) and informs the user. Then, this module will be responsible for submitting each new simulation to the Cloud.
- The Data Manager: This component uploads the input data and downloads the corresponding results and meta-data using a CDMI service or directly the Windows Azure Storage service.
- The Notification Manager: It obtains the changes in the simulation status and updates the associated information.

Every time the Remote Simulation Manager Client is launched, the user must be authenticated in the Cloud by means of the username and the password. If the user has permission to use the Cloud Service, i.e. he has been registered in the users table, then a list of his remote simulations will be shown, and he will be able to submit new simulation jobs, download results, cancel executions, and so on. For each simulation, geometric information related to the structure analysed, the type of analysis, the kind of data to be retrieved, the way of downloaded the results, etc. is exposed, including the current status, which is updated every time with a configurable frequency by the Notification Manager.

When the user submits a simulation to be executed in the Cloud, the following steps take place: First of all, the Architrave Analysis Component writes a binary file containing the building geometry, the external loads applied (such as an earthquake) and the simulation parameters, and sends a message containing the file path to the Remote Simulation Manager Client. Then, the Remote Simulation Manager Client reads the input binary file and the remote simulation is registered. Next, the Data Manager uploads the input binary file using the CDMI service to the CS. Finally, the Job Submission Manager submits the job to the Cloud Service in the way of a .jsdl document and the Notification Manager consults periodically the significant status changes in the Cloud Service. When the results are ready, the Data Manager will download them also by means of the CDMI service.

The user can configure the amount of results to be generated in the Cloud and to be received in the local machine, and the way to be downloaded. Thus, the Data Manager can be demanded to retrieve the simulation results automatically (when they are available) or manually (when the user requests them by clicking on the corresponding button). Also, in a dynamic analysis, the user can save on the CS the simulation results for all the time steps; the simulation results of just the 3 most unfavourable time steps according to the X, Y and module base shears or, finally, a video with the graphical response of the structure along the time. Besides, the user can configure the amount of data to be downloaded, obviously depending on the results previously computed and stored in the Cloud. In any case, a list with the X, Y and module base shears for all time steps will be always downloaded in order to allow the user to retrieve subsequently the amount of unfavourable and most significant simulation time step results that he desires.

Moreover, the Remote Simulation Manager Client can indicate the Data Manager to remove the input and output simulation data automatically (when the data results have been successfully received) or manually (when the user desires), to cancel the execution of a structural analysis or cancel the retrieval of the results.

3.2 The Structural Analysis Cloud Service

On the Cloud-side, the Structural Analysis Cloud Service is composed of the following components:

- The GW Service, i.e. a web-role implementation for Windows Azure that manages the execution of the remote tasks. This Service is composed of the Security Token Service (to perform the user authentication and allow that just authorized users can use it), the Scaling Service (to increase or decrease dynamically the number of worker instances), the Notification Service (to inform the user about the status of the jobs), the Submission Service (to send the simulations to the worker instances) and the Job Management (to provide information related to the status of the tasks).
- The Structural Analysis Service Manager, which generates the input files needed by the Structural Simulator and launches it and encapsulates the results.
- The Structural Simulator Worker, i.e. the HPC Architrave module that runs the simulations in the Cloud.
- The CDMI Service, which uploads and downloads the data using the standard CDMI protocol.
- The Elasticity Manager, which sets automatically the number of worker instances according to the workload.

For each simulation request, the associated input binary file is moved from the CS to the local drive of the VM by the GW Service, and the Structural Analysis Service Manager is launched. This executable file reads the input archive and generates the input file needed by the Structural Simulator Worker. Next, the Structural Simulator Worker is executed with the appropriate parameters, depending on the type of analysis, defined by the user. As a result, the structure is analysed and the output files are periodically generated and saved on the local drive. For each simulation time step, in case of a dynamic analysis with response along the time, all the multiple output files generated are encapsulated in just a single output file and sent to the CS during the execution by the Structural Analysis Service Manager.

A notification scheme is employed, where the Notification Service of the GW puts the status changes of the simulations into an Azure queue, which is consulted periodically by the Notification Manager component of the Remote Simulation Manager Client. In this way, the Remote Simulation Manager Client informs properly the user and downloads the output files to the local machine when the execution has finished, or when each simulation time step is ready to be retrieved, following a data-driven model. As a consequence, the remote simulation execution and the result retrieval phase are overlapped in time and, thereby, the whole time involved in the simulation is dramatically reduced.

The Structural Analysis Cloud Service contains an Elasticity Manager component able to adjust the number of available worker resources through the Scaling Service of the GW, depending on the workload in the system. Specifically, this component monitors continually the number of workers in execution, and a new instance worker is launched automatically when the processing workload is increasing. For that, when the number of instances waiting for jobs is lower than two, a new worker is deployed. On the other hand, if all the worker instances are idle, most of them will be terminated and the number of workers will reduced to the minimum.

4 Execution results and performance analysis

This section explains the representative case study applied to validate the Cloud system. The following subsections describe the structural and computational interest of the case study and the expected improvement benefits that provides the Cloud Computing solution.

4.1 Introduction to the case study

For the case study, we have selected a structure (Figure 3) corresponding to a reinterpretation, according to the current usages and regulations, of the original structure of the Nordic Bank, located in Helsinki. This is a work of the Finnish architect Alvar Aalto in 1962. It consists on a portal frame structure, solved by means of slabs over interior concrete columns and steel columns at the facade. The core of the vertical communication is materialized as reinforced concrete walls serving as vertical structure and lateral bracing. The spans are moderated, except in some zones at lowest levels where the foundation lab is

reinforced with hanging concrete beams. The facade steel columns are disposed to half spans, with beams of diversion at the level of first floor. The foundation consists of a slab, due to the presence of the phreatic stratum. The model has been designed with bars for columns and beams, and 2D medium sized finite elements for slabs and walls. For the slabs, a Delaunay mesh has been employed, while the walls are solved with a simple mesh.

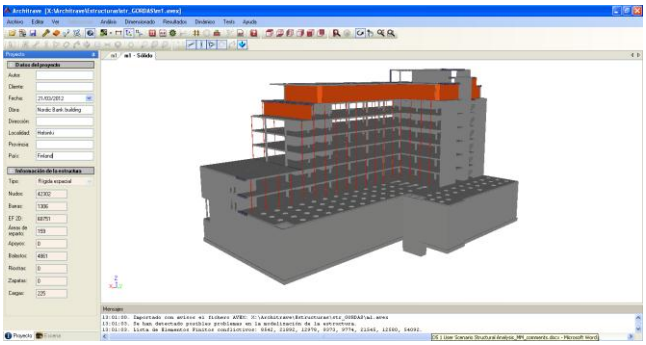


Fig. 3. Nordic Bank building.

The case study presented consisted on testing and simulating 10 different structural solutions that came from the same structural design, composed of 253812 degrees of freedom, with 1306 columns and beams, and 68751 2D finite elements. Each structural solution was composed of a variation of the column dimensions, and the slab and wall thickness. Moreover, each of them was dynamically tested under the influence of 5 representative earthquakes of 12 seconds of duration, with a simulation time increment of 0.01 seconds. These results were stored every 0.05 seconds. In this way, 1200 times steps were simulated and 240 of them were saved on disk. It should be noted that the aim of this case study was to select the best one of the 10 structural solutions that accomplished the structural and safety regulations and presented the most economic final cost.

Therefore, for the case study execution, a set of 50 independent 3D dynamic simulations were launched and computed, firstly locally in a conventional PC (Intel core i5, CPU@ 3.20 GHz and 4 Gbytes of RAM), and then in the Cloud, following two different configurations. On the configuration A, all the simulation time steps remotely saved were retrieved by the client; and on the configuration B, only the 3 most unfavourable time steps were saved and retrieved by the user. Furthermore, the two configurations were tested over different deployments composed of 1, 10, 25 and 50 medium-sized Web role Azure instances (CPU@ 1.60 GHz and 3.5 Gbytes of RAM).

4.2 Performance analysis

The validation tests demonstrate the behaviour of the VENUS-C platform according to the case study described in the previous section. For that, a quantitative evaluation has been carried out, measuring the efficiency of the platform by means of the response time and the speed-up, obtained by the Cloud approach with respect to the local approach and with respect to employ just one Azure instance in the Cloud.

Before executing the whole case study, the simulation of one of these structural solutions was computed firstly on premises and then in the Cloud. The Table 1 shows the results when computing the structural solution in the local machine of the user, together with the response time, the size of the input data and the amount of data downloaded by the local client, for the two described configurations.

Table 1: Results corresponding to the simulation of one of the structural solutions in the Cloud.

Type of execution	Response time (minutes)	Input data (Mbytes)	Output data (Gbytes)
Local execution - Saving the results of the 240 time steps	120.03	5.43	2.48
Configuration A	279.48	5.43	2.48
Configuration B	277.46	5.43	0.12

It should be clear that, for the configuration A, all the needed data movements (from the VM local disk to the CS and from the CS to the user machine) were overlapped with the execution. However, for the configuration B, the data transference just could start once the simulation had finished and the most adverse results had been computed.

The results of the Table 1 show a big difference between the local response time (120.03 minutes) and the remote response time (279.48 minutes for the configuration A, and 277.46 minutes for the configuration B).

For the itemization of the overhead times, the stages that give place to an overhead faced with the local execution have been analysed. Table 2 shows the time involved in each of the stages that compose the simulation of a single structural solution, where only the most unfavourable results are moved to the CS and downloaded later (configuration B). As it can be seen, the time involved in computation implies a delay of 155.28 minutes, with respect to the local execution.

Table 2: Execution times of each of the different stages involved in the simulation in the Cloud (configuration B).

	Data and job submission	Application download and job initialization	Structural Simulator Worker execution and result encapsulation	Result upload to the CS	Result download to the client machine
Local execution	-	-	120m 02s	-	-
Remote execution	6s	39s	275m 19s	50s	34s
Overhead time	6s	39s	155m 17s	50s	34s

The values of the Table 2 reflect that the most of the overhead time resides on the execution of the Structural

Simulator Worker, responsible of analysing the structure, and the Structural Analysis Service Manager, in charge of encapsulating the results generated. As the computational complexity is similar on the local and the Cloud simulation execution, it can be assumed that the main overhead resides on the difference on hardware characteristics, especially on the CPU features and the speed of accessing the disk for the read and write operations.

The overhead times at the configuration A shows a similar behaviour than in the case of the configuration B, as reflected in the Table 3. However, this is due to the fact that, as indicated above, the Structural Simulator Worker execution time is overlapped with the output data upload to the CS and with the result download to the client machine. In this way, the column 4 includes the time spent on the simulation execution, where the results of most of the time steps had been transferred simultaneously to the CS and received by the client. On the other hand, the column 5 shows the time required for the data upload to the CS and the result download to the client for the last time steps, i.e. once the Structural Simulator Worker execution has finished. Therefore, the main overhead appears during the simulation of the structure, with a delay of 154.88 minutes with respect to the local execution.

Table 3: Execution times of each of the different stages involved in the simulation in the Cloud (configuration A).

	Data and job submission	Application download and job initialization	Structural Simulator Worker execution and result encapsulation	Result upload to the CS and download to the client machine
Local execution	-	-	120m 02s	-
Remote execution	6s	39s	274m 55s	3m 49s
Overhead time	6s	39s	154m 53s	3m 49s

Once analysed the execution times of a single structural solution, the whole case study execution was performed, whose fifty different simulations were launched at the same time, and the status of the simulations were consulted periodically by means the Remote Simulation Manager Client. The evaluation compares the response time of the local execution, considering that each structural solution is analysed after another in the machine of the user, with the remote response time, for the two different previous configurations, over a set of different Cloud deployments composed of 1, 10, 25 and 50 medium-sized Azure instances. The response time was measured as the difference of time between the first job submission and the final result data downloaded corresponding to the last simulation.

The execution of the 50 simulations that compose the case study spent 100.03 CPU hours in a traditional approach, generating output results in the order of 124.04 Gbytes. In the Cloud deployment, the execution of the whole case use configuration A required 271.5 Mbytes as input data and 124.04 Gbytes of output data were produced. For the configuration B, 271.5 Mbytes were required as input data and 5.75 Gbytes of output data were generated.

Figure 4 shows the execution times of the whole case use, for both configurations and over a Cloud deployment composed of different number of Azure instances.

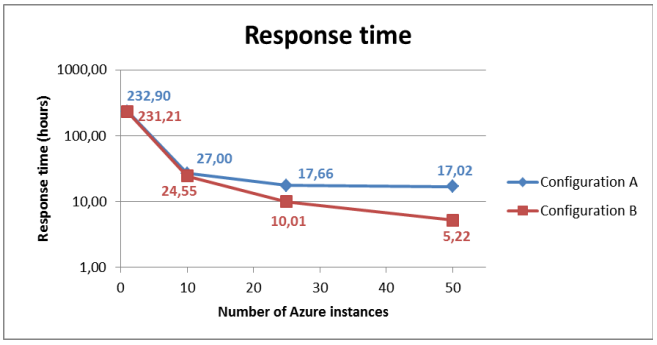


Fig. 4. Response time in hours for the whole case use.

As it can be noticed, the response time decreases gradually when the number of Azure instances is increased at the configuration B. In the case of the configuration A, the results show initially a similar trend. However, when the number of Azure instances that executes the case study is greater than 25, the results acquire a value of around 17 hours. This is due to the bottleneck of the network in the retrieval of this large volume of output data by the client, since the time required for downloading all the results generated is much greater than the time required for executing the simulations. Therefore, although the execution and the result retrieval stages are properly overlapped, the time required for the complete case study is mainly determined by the time spent on the reception of the 124.04 Gbytes of results generated.

In any case, it should be taken into account the dramatically reduction of the total time required for the execution. Whereas more than 4 days were needed in the traditional approach, just 17.02 and 5.22 hours were spent, respectively for the configurations A and B, when computing remotely all the simulations in the Cloud.

Figure 5 shows the behaviour of the Cloud system, when the number of Azure instances is increased, in terms of speed-up, with respect to the sequential approach (each structural solution analysed after another in the client local machine) and with regard to the results of the execution over just a single Azure instance. In this figure, it can be appreciated how the results of the speed-up, compared with the sequential approach, are much lower than results of the execution in an Azure instance, at the two configurations. This difference in the results is mainly due to the difference in the hardware characteristics between the local testing machine and the Azure instances.

Whereas the speed-up at the configuration B with regard to an Azure instance obtains values near to the ideal case (44.29 for 50 machines), the speed-up with respect to

the sequential approach is far from these values (19.16 when using 50 instances). In the case of the configuration A, it can be appreciated how the value of the speed-up does not increase appreciably when the number of Azure instances is greater than 25, due to the reasons exposed above related to the bottleneck of the network in the retrieval of the results.

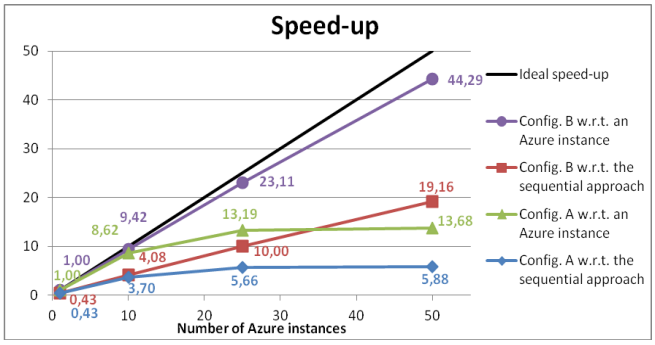


Fig. 5. Speed-up for the whole case use.

The behaviour, in terms of efficiency, of the Cloud system is reflected in the Figure 6, with respect to the sequential approach and with regard to an Azure instance.

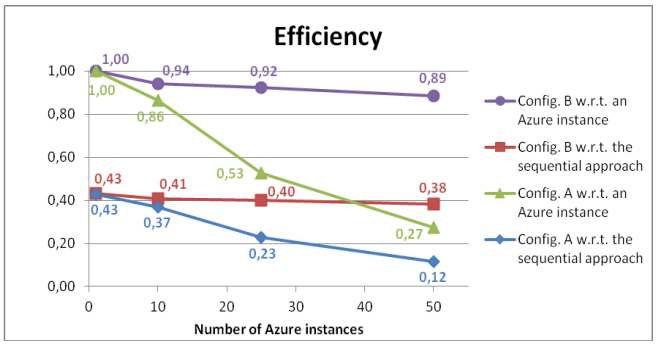


Fig. 6. Efficiency for the whole case use.

As expected, excellent efficiencies were obtained for the configuration B, but worse efficiencies were computed for the configuration A. Clearly, it can be noticed how the efficiency decrement is very low at the configuration B when the number of Azure instances is increased, whereas the tendency of the decrement at the configuration A is much more pronounced.

5 Conclusions

In this work, a Structural Analysis Cloud Service has been implemented, based on the GW component developed of the VENUS-C project, and deployed for Windows Azure. The software architecture of Cloud system is described, together with its design and the different elements that composed it.

All the remote simulations are managed by Architrave, an advanced software for structural analysis. The Architrave Analysis Component and the Architrave Structural Simulator have been properly adapted to work in the Cloud. In order to launch and manage the executions, the Remote Simulation Manager Client application has been implemented.

Thanks to the high throughput and reliable Structural Analysis Cloud Service implemented, researchers will have available a huge number of computational resources to be on-demand employed and lots of cost-effective simulations will be launched simultaneously. Thus, more experiments will be analysed per time unit, increasing the number of structures simulated and speeding up the research process.

The structural community will be able to solve larger scale problems, increase the complexity of the structure to be analysed, and carry out a larger number of realistic dynamic simulations. In this way, the reliability and safety of the results obtained will be improved and new structural problems will be tackled. Since the time spent on the design of buildings and civil engineering structures will be reduced, the engineering companies and the architectural studios will increase their productivity and volume of business.

Finally, there will be no need of acquiring software licenses in property and expensive hardware for solving large-scale structural problems (just pay per use), and the users will not be worried about new software updates.

6 References

[1] R.W Clough and J. Penzien. "Dynamics of structures". Second Edition. Computers and Structures, Inc., 2004.

[2] Architrave website: http://www.architrave.es

[3] VENUS-C project website: http://www.venus-c.eu

[4] D. Betts, et al. "Moving Applications to the Cloud on the Windows Azure Platform". Microsoft, 2012.

[5] I. Foster et. al. "OGSA Basic Execution Service Version 1.0". Grid Forum Document GFD-RP. 108. 8/8/2007.

[6] A. Savva. "Job Submission Description Language (JSDL) Specification". Version 1.0, GFD-R.056, 2005.

[7] SNIA CDMI: http://www.snia.org/tech_activities/standards/curr_standards/cdmi

[8] Y. Simmhan, C. van Ingen, G. Subramanian, and J. Li. "Bridging the Gap between the Cloud and an eScience Application Platform". Microsoft Research. U.S., 2010.

[9] Christian Geuer-Pollmann. "The VENUS-C Generic Worker". 1st annual SICS Cloud Day, 2011.

[10] D. Lezzi, et al. "COMPSs in the VENUS-C Platform: Enabling e-Science Applications on the Cloud". Proceedings of the IBERGRID 2011 Conference, Santander, 2011.

[11] I. Livenson, E. Laure. "Towards Transparent Integration of Heterogeneous Cloud Storage Platforms". Fourth International Workshop on Data Intensive Distributed Computing (DIDC 2011), San José, 2011.

Power Saving for Fast Deployment Private Cloud Toolkit – Ezilla with Infrastructure Services

Chang-Hsing Wu[1], Yi-Lun Pan[1], Hsi-En Yu[1], Hui-Shan Chen[1] and Weicheng Huang[1]
National Center for High-performance Computing, Taiwan, R.O.C.
{hsing, serenapan, yun, chwhs, whuang}@nchc.narl.org.tw

Abstract – *As virtualization technologies become more prevalent, Cloud users usually encounter the problem of how to build his/her own virtual cluster form cloud environment. We provided easy installation toolkit for building cloud environment with friendly user interface. Ezilla toolkit has been developed by the Pervasive Computing Team at the National Center for High-Performance Computing (NCHC). Ezilla toolkit integrates the de facto Cloud middleware, Web-based Operating System (WebOS), and coordinated Cloud infrastructure services (hypervisor, storage, and networking) to form a cloud environment. With a click, Cloud users can easily customize and configure the specified cloud environment via Ezilla toolkit. The main feature of Ezilla is simplifying a lot complexity of utilizing Clouds. Our goal is to make scientists or users painlessly run their works on Clouds.*

Keywords: Virtualization Techniques; WebOS; Virtual Cluster.

1 Introduction

Cloud users have to manually build specified virtual cluster of cloud environment with the console mode in order to manage or generate virtual resources. To improve this condition, an *Ezilla* toolkit has been developed by the Pervasive Computing (PerComp) Team at the National Center for High-Performance Computing (NCHC). Ezilla is built on the "Carry-On-Cloud" concept. On this platform, Cloud users can build on demand virtual clusters with one click. Furthermore, Ezilla leverages unattended installation technique, Cloud middleware, WebOS (Web-based Operating System), and DRBL - SSI mode (Diskless Remote Boot in Linux - Single System Image) [1], to provide the software infrastructure of the Ezilla system. Therefore, Cloud users can build the whole Cloud environment very easily.

Ezilla toolkit provides a user-friendly interface that is WebOS. The WebOS infrastructure offers a seamless and unified access to geographical distributed resources connected via Internet, and it can supply most basic operating system services [2]. Thus, with one click, Ezilla toolkit helps Cloud users to make their own private Cloud easily. The progress of Ezilla helps to lower the barrier for using Cloud computing environment. The designed Ezilla toolkit has become necessary to provide Cloud users with an interface that is both user-friendly and straightforward. This research focuses on virtual resources management with an interactive graphical user environment.

Furthermore, the "Green Computing," is especially important and timely. As Cloud Computing becomes increasingly pervasive, the energy consumption attributable to computing is climbing, despite the clarion call to action to reduce consumption and reverse greenhouse effects. To echo today's energy saving issues, we also developed a power saving approach in Ezilla toolkit to reduce energy utilization in Cloud Computing resources. We do this work on the integration of local scheduler that aims at reducing power consumption such that they suffice for meeting the minimizing quality of service required by local cluster or physical machines.

According to the above scenario, those issues encourage the motivation of our research and development. Specifically, the design and implementation of the proposed Ezilla toolkit includes Cloud middleware, WebOS, DRBL – SSI mode, and power saving mechanism. The rest of the paper is organized as follows. Section 2 presents related works, including existing Web-based Operating system, virtualization technologies, and Diskless Remote Boot in Linux (DRBL). In Sections 3, we propose the implementation of Ezilla toolkit and power saving Mechanism that we developed. In Section 4, the research results. The conclusion and future directions are presented in Section 5.

2 Related Works

2.1 Existing Web-based Operating System (WebOS) Projects

Recently, a famous WebOS - Chrome OS, developed based on AJAX technique [3]. It can be used to implement a web application that communicates with a server in the background, without interfering with the current state of the page. The developments of Cloud WebOS platform via AJAX technique become practicable. However, Chrome OS does not provide on-demand applications and computing services to users in Clouds.

A Web-based Operating System (WebOS) project started as part of Network of Workstations [4] at the University of California, Berkeley in 1996. So far, there are several typical commercial projects of WebOS, such as FlyakiteOSX [5], Glide OS [6], Xindesktop [7], … etc. All of these systems are online OS with AJAX and PHP

techniques adopted. However, these projects are not open source and lack of the management of distributed computing resources. To tackle the technical issues of distributed computing resource management, the Cloud WebOS platform is developed in this work, upholding the spirit of open source, open standard, and is distributed under GNU/GPL license.

2.2 Virtualization Technologies

The virtualization technology is not a brand new technology. In the late 1990s, virtualization was achieved by complex software (binary translation) techniques given the fact that the virtualization technique was not supported by processors at the moment and obtained reasonable performance. In the late 2006, both the Intel and the AMD created new processor extensions to their processors and enhanced the support of the virtualization. Furthermore, they also implemented the I/O virtualization technology that covers memory, disk and network by the chipset. The technology was called hardware-assisted virtualization. This approach increases the performance by removing a layer of complex software techniques between the guest OS and hardware. Examples of virtualization platforms that are adapted to such hardware include Linux KVM, VMware Workstation, Microsoft Hyper-V, to name a few.

In the arena of virtualization technology, hypervisor is considered as a crucial part of the technology. It is software that manages multiple virtual machines on a single computer system. Hypervisor is a layer resides between the existing operating system and hardware to manage the computing hardware and virtual machines.

Generally, modern implementations of hypervisor are divided into two categories, including Host-based and Bare-metal approaches. The host-based approach uses modified operating systems to provide virtual machine monitoring, such as Linux-VServer [8], Solaris Zones [9], and Kernel-based Virtual Machine (KVM) [10]. On the other hand, the bare-metal approach employs small-dedicated hypervisors to directly run on physical machines. The VMware ESX server [11] and the XenServer [12] are the famous examples of the bare-metal approach.

2.3 Diskless Remote Boot in Linux (DRBL)

Diskless remote boot in Linux (DRBL) environment is based on the network boot mechanism, network file system, and centralized authentication mechanism. To boot the computing node remotely, a mechanism that can download program over network is required. There are many programs fulfill this need, such as Preboot Execution Environment (PXE) [13], etherboot [14], gPXE [15], and iPXE [16]. The DRBL is compatible with all of them. Once the DRBL server is ready, the whole computing cluster is ready. It is not necessary to install the OS onto computing nodes at all. Services, such as dhcp, tftp, NFS, and NIS are executed in the DRBL server to serve all the computing nodes.

By setting computing nodes to boot from network, IP addresses will be assigned from server when nodes during the booting procedure. Once a computing node gets its IP address and kernel, it will boot from the kernel and the initialized RAM disk "initrd". The computing node will then load the driver for NIC in the initrd, mount its root directory and other necessary directories via NFS. Once the procedure is completed, the computing node has all the necessary files. Then, the normal boot procedure, just like that of regular Linux boot steps in the diskfull nodes, will follow. Standardized procedure and packages needed to setup a DRBL server has been developed for various Linux distributions, including Debian, Red Hat, Fedora, and Mandrake. The whole setup procedure is easy, straightforward and productive. Following the documentation on the DRBL project website, one can setup a DRBL server in about half an hour, depending on the packages chosen to be installed and the network speed.

To boost the performance of a DRBL cluster, performance tuning in the DRBL setup program was carried out. One of the examples is that, there are more than 100 device files under the directory /dev in a diskless node. If these small but numerous devices files exist in the NFS server and share with the clients via NFS, the cluster performance won't be good. So the tmpfs file system in the diskless clients is used and these device files are created in this tmpfs file system when diskless clients boot. With most of these tmpfs files systems for directories /etc/, /var, /tmp/, and /dev/ included in the DRBL clients, which is called Single System Image (SSI) mod, loading of the file server is greatly reduced.

With success of the virtual technologies, we integrate virtualization technology – KVM, WebOS, and DRBL. This research comes up with a new and lightweight approach to access virtual computing services via the Ezilla toolkit.

3 The Implementation of Fast Deployment Ezilla Toolkit and Power Saving Mechanism

3.1 Research Objective

The key idea of Cloud Computing lies in its component-based nature, which are reusability, substitutability and user friendly. By integrating virtualization technologies, DRBL – SSI mode, and WebOS, a web environment to access Cloud services via Ezilla WebOS Interface is provided. This progress helps to lower the barrier for using Clouds. This research focuses on virtual resources management with an interactive graphical user environment. More specifically, the Ezilla toolkit implements an autonomic virtual computing resources management system based on decentralized resource

discovery architecture. At the same time, a power saving mechanism is also crucial to the utilization of Cloud resource. Therefore, the designed power saving mechanism is used for the virtual cluster and physical cluster.

3.2 Implementation of Fast Deployment Ezilla Toolkit

There are three components in the Ezilla toolkit, including Ezilla Server, Ezilla Client and intuitive Ezilla WebOS Interface, as sketched in the Figure 1. The Ezilla Server responses for orchestrating computing resources and monitors the status both of physical and virtual machines via cloud middleware. In addition, the most important feature is the image file server used to manage virtual machine images. The VM images are stored in the file system that controlled by the Ezilla server. Responding to the request from Cloud users, the images will be dispatched to Ezilla clients and used to generate virtual machines in turn by Ezilla clients. A Cloud user can use his personal Ezilla WebOS interface to create a Cloud environment linking storage, networking, and computing power timely and easily.

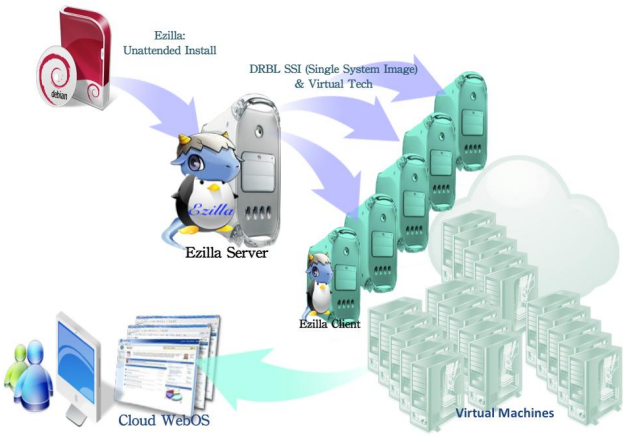


Figure 1. The System Architecture of Ezilla

The Cloud users can create dynamically a virtual cluster consisting of VMs through Ezilla WebOS Interface. Users determine the scale and the life span of the virtual cluster. Once the virtual cluster is no longer needed, it can be dismissed subjected to the decision of the user. The computing resource is then released for other users to use. Due to the adoption of the XML-RPC based API for the development of the WebOS; the whole operation can be manipulated via a web browser, such as Chrome. There are several middlewares embedded in the designed Ezilla Toolkit:

- Integration with basic Operation System and packages – The Ezilla toolkit supports the Debian OS and basic packages, including KVM hypervisor, and Libvirt [17]. In order to improve the performance, the Virtio [18] driver is also adopted in the virtual machines created by the Ezilla. Virtio driver provides paravirtualized functions for network virtualization and disk I/O virtualization. On the scenario of Network, the Bridge mode is configured by Ezilla toolkit as well.

- Integration with *OpenNebula* [19] – OpenNebula is used as central cloud management. It is responsible for allocating available computing resources, creating VMs based on user selected VM image, and deploying the image onto the physical computing resources. It also manages unique MAC address, IP address, and virtual network (vNet) ID. Therefore, each user's virtual cluster lives on its own vNet without interfering with other virtual clusters.

Once the Ezilla Server is in place, Ezilla Clients can be setup automatically via the DRBL–SSI Mode and existing virtualization technologies. With such a mechanism implemented, a physical machine can be easily turned into Ezilla Client via PXE (Preboot Execution Environment) with the local disk untouched. Hence, computing resources can be added dynamically without any reconfiguration, thus to enhance the flexibility of Cloud resource allocation.

3.3 Power Saving Mechanism

We developed an approach to reduce energy utilization in Ezilla cluster. We do this work on the integration of OpenNebula and remote power management system that aims at reducing power consumption such that they suffice for meeting the minimizing quality of service required by cluster. In particular, our approach relies on recalling services dynamically onto appropriate amount of the Ezilla Clients according to user's job request (virtual machines request) and temporarily shutting down the Ezilla Clients after finish in order to conserve energy.

As shown in Figure 6, the power saving mechanism will wake up every minute to check job (virtual machines request) queue if there exist jobs, and make sure Ezilla Client become available, the power saving mechanism then will fetch the applicable jobs, parses the requirements, and remotely powers on the correct number of machines by Wake-on-LAN [20] protocol or IPMI [21]. After the job completes, the power saving mechanism powers the machines down. Our implementation currently relies on checking the scheduler's job pool and then decides to shut down which Ezilla Client when no new job was submitted. By powering off idle Ezilla Client, it can significantly save more energy than always keeping all Ezilla clients running.

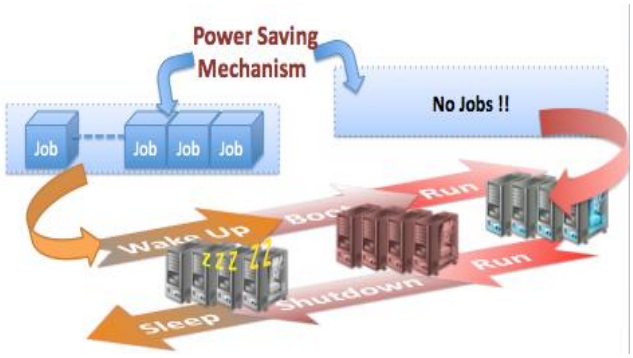


Figure 2. Scenario of Power Saving Mechanism

4 Research Results

4.1 Fast Deployment of the Ezilla Toolkit

The installation of the Ezilla was designed to be as effortless as possible so that a Cloud environment can be deployed quickly and easily. In fact, an Ezilla Server can be deployed in just three simple steps via DRBL – SSI mode, as shown in the following Figure 3, 4, and 5.

- Step1 – Boot the server with a genuine Debian CD-ROM and select "Help", for example, in Figure 3.

Figure 3. Step1 - boot the server with a genuine Debian CD-ROM and select "Help"

- Step2 – Key in "auto url=ezilla-nchc.sf.net" once the system enter the "Help Index", in Figure 4. The installation of the server is fully automated with all the needed packages and source codes downloaded.

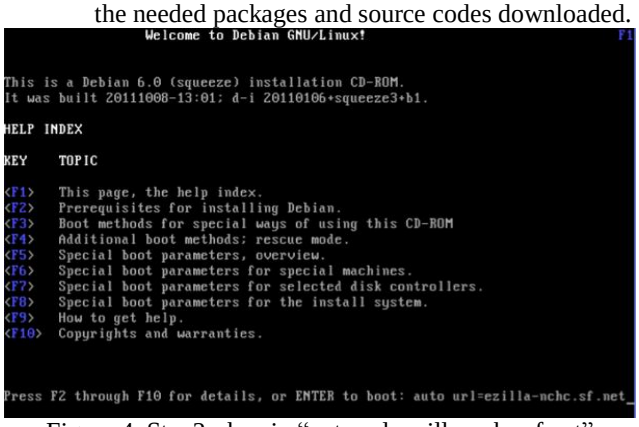


Figure 4. Step2 - key in "auto url=ezilla-nchc.sf.net"

- Step3 - Per the illustration in the Figure 5, the Local Area Network requires static IP instead of the DHCP.

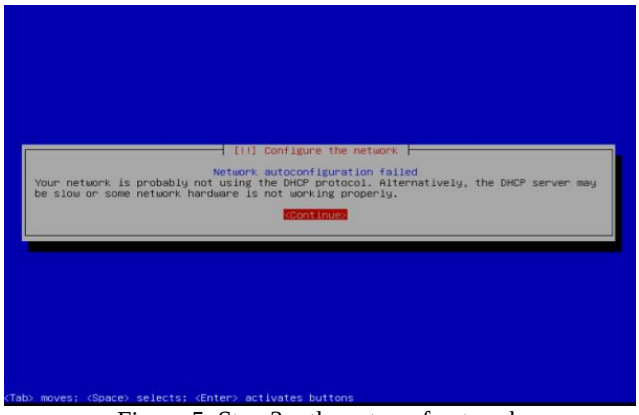


Figure 5. Step 3 – the setup of network

After completion of all the three steps above when the Ezilla server is ready for further use, the automatic installation process for building the Ezilla environment takes over. The installation will proceed without requiring the attention from system admin. The installation is completed and the Ezilla Cloud environment is ready for use once the boot menu page is shown, as in the Figure 6.

Figure 6. The boot menu on Ezilla Client with PXE

4.2 Ezilla WebOS Interface and Widgets

More advanced Cloud Computing Widgets are attempted as well. One of the most important results in this paper is that we have developed many Cloud Widgets with friendly graphical user interface via Ezilla WebOS Interface. The kernel of this system architecture consists of four Widgets, including Image Creator Widget, VM Creator Widget, VM Monitor Widget, and VM Control Widget. Users without much learning effort can easily manage all of these widgets. These Widgets allow users to customize and to arrange their complicated computing tasks according to their requirements.

The Image Creator Widget, in the figure 7, is to generate the customized base image and on-demand/specified application from the end users' requirements. This Widget provides a complete and integrated HPC software stack that

consists of operating system, management tools, resource monitor, and even commercial package, such as the Matlab. VM Creator Widget - with the profile of virtual cluster demanded by the user provided, it will generate a specification, shown in the figure 8, which in turn is parsed by the VM Creator engine to create an on-demand virtual Cluster on the physical computing resources.

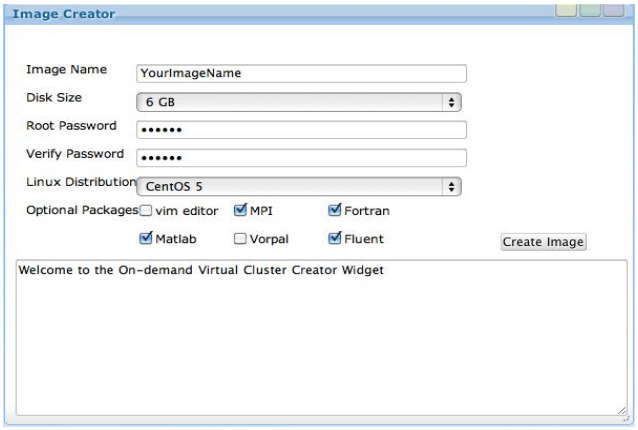


Figure 7. Image Creator Widget

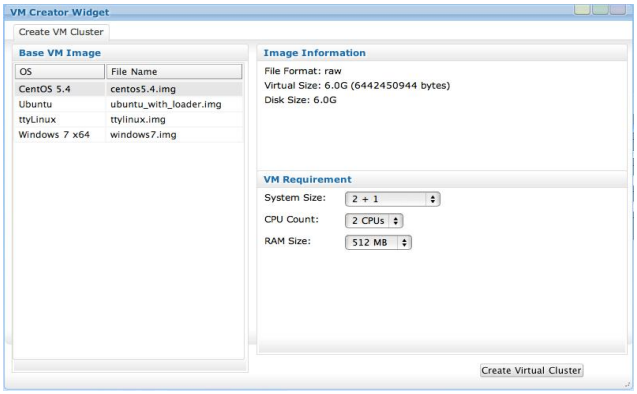


Figure 8. VM Creator Widget

In the figure 9, the main task of the VM Monitor Widget is to monitor the all the status of virtual machines, Networks, and the physical hardware. In addition, this Widget makes use of the information and the status provided by the Monitoring & Reporting Cloud Middleware. The VM Control Widget is designed for such a purpose with Cloud visualizer integrated as the core of its Cloud service, as shown in the figure 10.

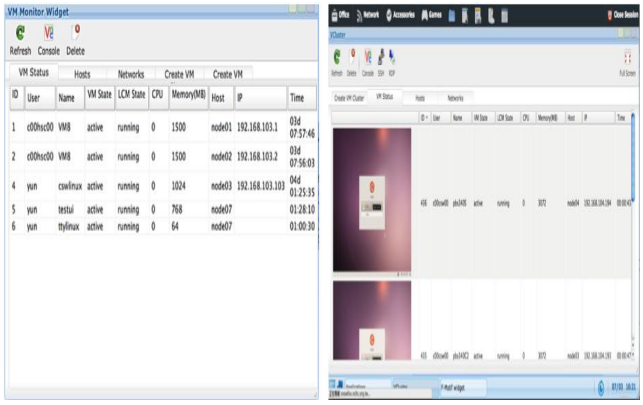


Figure 9. VM Monitor Widget

Figure 10. VM Control Widget Cloud Visualizer – Windows 7 Booting Status

Moreover, we used Ezilla toolkit to implement the following two customized applications for biological simulation and information security simulation in Eziila WebOS. The F-motif Simulation Widget provides specialized Cloud services to search and analyze the sequence of gene in real time, in figure 11. The other customized Cloud Widget about information security is called ICAS (IDS-log Cloud Analysis System) Widget. As long as user selects the ICAS base image, the Hadoop DB and virtual cluster are constructed automatically, and then users can analyze the IDS-log as the figure 12 shown.

In the figure 13 and 14, Ezilla WebOS also provides two methods to control virtual machines. One is VNC, the other is SSH on Web.

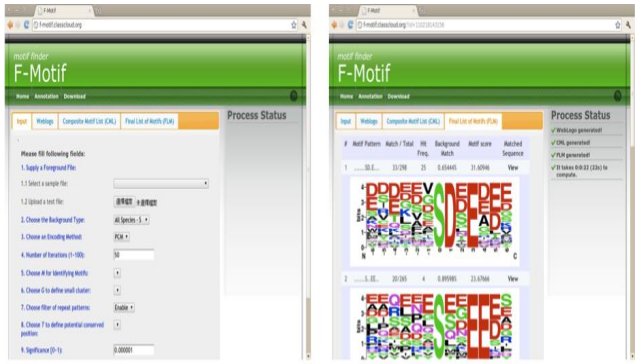


Figure 11. F-motif Widget in Ezilla WebOS

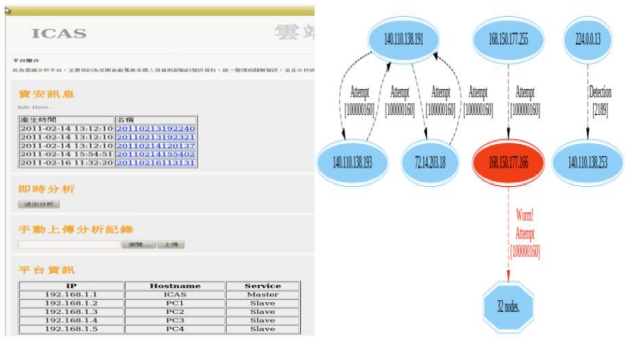


Figure 12. ICAS Widget in Eziila WebOS

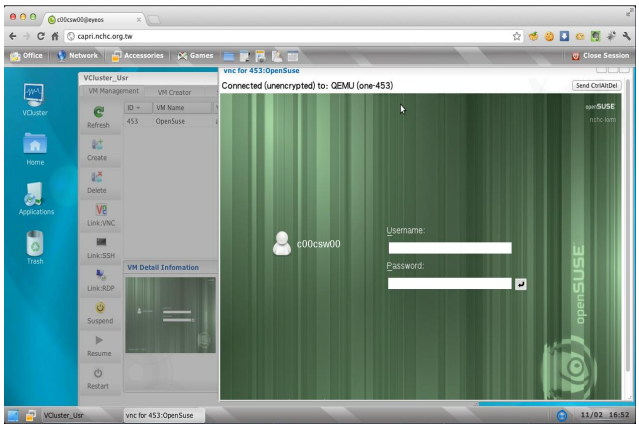


Figure 13. Using VNC to Control VM

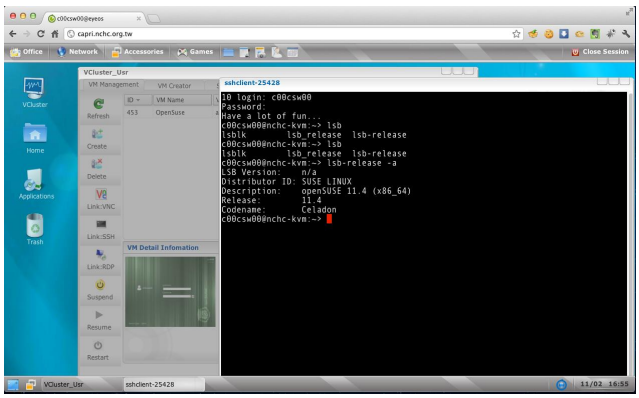


Figure 14. Using SSH to Control VM

5 Conclusions Conclusion and Future Work

The proposed toolkit – Ezilla [22], it not only helps user to build virtual cluster easily and automatically, but also provides different varieties of computing environment such as Linux, Win7, and so on. Furthermore, the ability to distribute and balance the workload across multiple physical as well as virtual computing resources will be tackled in the future development of this research.

The NCHC's Ezilla Development Team hopes that others will use Ezilla to create diverse applications, which incorporate additional Cloud services that can be accessed anytime. The ultimate goal of Ezilla is to achieve "Every as a Service" usability.

6 References

[1] DRBL 2012, URL: http://drbl.org

[2] G. Gu, and X. Lu, "Simple Web OS System Based on Ext Framework and Cloud Computing," *International Forum on Information Technology and Applications*, pp. 448-450, IEEE, 2010.

[3] http://www.chromium.org/chromium-os/, 2011.

[4] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A, Patterson, "Searching for the Sorting Record: Experiences in Tuning NOW-Sort," *The 1998 Symposium on Parallel and Distributed Tools (SPDT '98)*, Welches, Oregon, August 3-4, 1998.

[5] http://osx.portraitofakite.com/logon.htm, 2011.

[6] http://www.glidedigital.com/, 2011.

[7] http://www.xindesk.com/, 2011.

[8] S. Soltesz, H. P¨otzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors, " *Proceedings of ACM SIGOPS/Eurosys European Conf. on Computer Systems*, pp. 275-287, Mar. 2007.

[9] D. Price and A. Tucker, "Solaris Zones: Operating Systems Support for Consolidating Commercial Workloads, " *Proceedings of 18th Large Installation System Administration Conf.*, pp. 241-254, Nov. 2004.

[10] B. Zhang, X. Wang, R. Lai, Y. Liang, Z. Wang, Y. Luo, and X. Li, "Evaluating and Optimizing I/O Virtualization in Kernel-based Virtual Machine (KVM)," *International Conference on Network and Parallel Computing*, pp. 220-231, Zhengzhou, China, September 13-15, 2010.

[11] John Paul, "VMWare ESX Server Workload Analysis: How to Determine Good Candidates for Virtualization," *33rd International Computer Measurement Group Conference*, pp. 483-484, San Diego, CA, USA, December 2-7, 2007.

[12] X. Ge, H. Jin; S. Wu, X. Shi, W. Gao, "A method of multi-VM automatic network configuration," *Intelligent Computing and Intelligent Systems*, pp. 309-313, 2009.

[13] PXE 2011, URL: http://www.intel.com/support/network/adapter/pro100/bootagent/sb/cs-008191.htm

[14] Etherboot 2011, URL: http://etherboot.org

[15] gPXE 2011, URL: http://etherboot.org

[16] iPXE 2011, URL: http://ipxe.org

[17] Bolte, M., Sievers M., Birkenheuer, G., Niehorster O., and Brinkmann A. (2010) 'Non-intrusive Virtualization Management using libvirt', *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, Germany, pp. 574-579.

[18] Virtio 2011, URL: http://www.linux-kvm.org/page/Virtio

[19] Milojičić, D., Llorente, I.M., Montero, R.S, 'OpenNebula: A Cloud Management Tool', *Internet Computing, IEEE*, pp. 11-14, 2011.

[20] Wake-on-LAN 2011, http://en.wikipedia.org/wiki/Wake-on-LAN

[21] IPMI 2011, http://www.intel.com/design/servers/ipmi/index.htm

[22] Ezilla 2012, http://ezilla.info

Resource Assignment in Computational Grid Based on Grid Market Equilibrium

Xi XIE[1] **and Satoshi FUJITA**[1]
[1]Department of Information Engineering, Hiroshima University
Higashi-Hiroshima, 739-8527, Japan

Abstract— *In this paper, we propose a resource assignment scheme in the computational grid based on the notion of market equilibrium. Market equilibrium is a key concept commonly used in the field of game theory, and in this framework, we determine the "price" of resources owned by the service providers so that it fulfills the Nash equilibrium of the given market consisting of clients and service providers. The degree of satisfaction of clients is modeled as a linear utility function of acquired resources, and as a constraint concerned with the clients, we use the notion of budgets. Our proposed scheme a semi-algorithm which finds an assignment of resources to the clients so that it maximizes the utility of the clients provided that: 1) all resources are completely exhausted and 2) the surplus of the clients is at most ϵ using $O(n \log(nM/\epsilon))$ maximum flow computations, where M is the total amount of budgets given to the clients at the initial state and ϵ is a positive real representing the accuracy of approximation.*

1. Introduction

1.1 Background

Our daily life is being enriched with the aid of advanced distributed systems such as cloud computing (e.g., Google Apps and Amazon EC2), social networking services (e.g., Twitter and Facebook), and on-line streaming services (e.g., YouTube and Ustream) which have a high computational availability compared with classical distributed systems with respect to the performance, reliability, usability, and the fault-tolerance. As the availability of such advanced systems increases, however, it has raised another crucial issue so that how to utilize miscellaneous resources distributed over the network in an efficient and productive manner. In fact, the resource management has been recognized as a key issue to realize an efficient utilization of fully distributed systems such as volunteer computing and peer-to-peer systems, and in the past three decades, a huge number of resource management schemes have been proposed (see [11], [3] for survey).

Computational grid, which is often called Grid, is an emerging technology to improve the utilization of shared resources in large-scale distributed systems. Grid has been used to solve many important problems in the field of business, finance, medicine, and Grand Challenge applications in many fields of science and technology. In addition, there are several national projects concerned with the development and the utilization of Grid, e.g., Fusion Collaboratory Project in United States [14], EU DataGrid in EU [15], China National Grid in China [12], NAREGI in Japan [13], and others. The reader should note that the resource management in Grid should satisfy the requirement from both of "clients" and "service providers." More specifically, from the client side, it is required that spending the least money and getting the highest satisfaction. On the other hand, from the service provider side, it is required to make a full utilization of given resources.

There are many resource management schemes proposed in the literature, and some of them could certainly improve the *efficiency of resource utilization* in Grid. Agent-based scheme [7] and reputation-based scheme [1] are two representatives in this direction. A resource management based on the notion of economics, which is referred to as Grid Economy, has also received considerable attention in the past few years [16], [8]. The Grid Economy provides mechanisms to trade-off QoS parameters, deadline and computational cost, and offers an incentive for relaxing their requirements. However, most of those conventional schemes are based on a classical model of distributed systems such that *resources should be assigned to clients so that a certain cost function is optimized subject to a certain constraint*. In other words, it defines the problem from the viewpoint of the system manager and ignores the "emotion" of individual clients (clients) such as incentive, satisfaction, and regret.

1.2 Our Contribution

To overcome such a critical problem existing in conventional schemes, in this paper, we adopt the notion of *market equilibrium* to satisfy requirements issued by clients and service providers. Market equilibrium is a key concept commonly used in the field of game theory. In this framework, we will determine the "price" of each resource owned by the service providers so that it fulfills the Nash equilibrium of the given market consisting of clients and service providers. More concretely, we assume the existence of a centralized manager, and instead of determining the price of resources using an autonomous (but one-sided) mechanism such as buyer initiated or seller initiated auction,

the price of resources will be determined by the centralized manager so that:

1) the satisfaction of each client is maximized, and
2) given resources are fully utilized.

The degree of satisfaction is modeled as a linear utility function which is independently defined for each client, and as a constraint concerned with the clients, we use the notion of budget in the price determination process.

As a concrete method to calculate such a market equilibrium, we focus on a polynomial time algorithm proposed by Devanur *et al.* in 2002 [9]. This algorithm calculates a market equilibrium using $O(n^4(\log n + n \log U + \log M))$ max-flow (maximum flow) computations, where n is the number of service providers, U is the maximum coefficient in the utility functions and M is the total money possessed by the clients at the initial state (the details of the algorithm will be described in Section 4). Thus, although it is in fact polynomial time and always outputs an exact solution, it rapidly glows as the size of the given instance increases. In this paper, we reduce the time complexity of Devanur *et al.*'s algorithm using the notion of *approximation*. More precisely, our proposed scheme is a *semi-algorithm* which calculates the price of resources maximizing the utility of clients provided that: 1) all resources are completely exhausted and 2) the surplus of the clients is at most ϵ for any $\epsilon > 0$, using $O(n \log(nM/\epsilon))$ max-flow computations. The reader should note that our scheme solves (slightly) different problem from the original problem in the sense that we allow the remaining of very small surplus (of at most ϵ) whereas such surplus is strictly prohibited in the original problem.

The performance of the proposed scheme is experimentally evaluated with respect to the following factors: 1) the percentage of instances successfully solved by the proposed scheme (recall that our scheme is not an algorithm but a semi-algorithm), 2) comparison of the execution time with the Devanur *et al.*'s algorithm, and 3) certification on the approximation, i.e., whether or not the surplus in the resultant solution is certainly smaller than given ϵ.

The reminder of this paper is organized as follows. Section 2 overviews related work. Section 3 defines a service model of Grid adopted in this paper. Section 4 outlines the primal-dual algorithm proposed by Devanur *et al.* Section 5 describes our proposed scheme. Section 6 shows the result of experiments. Finally, Section 7 concludes the paper with topics for future work.

2. Related Work

This section overviews several resource management schemes proposed in the literature which can be applied to the resource management in the Grid.

2.1 Economic Model

In order to encourage the incentive of the clients, Buyya *et al.* introduced the notion of economy to the grid environment [6]. In their model, the manager can increase the degree of satisfaction of the clients by allocating resources to the clients through an auction mechanism. The auction mechanism supports one-to-many negotiation between a service provider and a set of clients, and will derive a single value (i.e. price) as an outcome of the negotiation. The manager regulates rules of the auction, which should be acceptable for both of clients and service providers, and uses the "force from the market" so that the negotiation should derive a clearing price for the service.

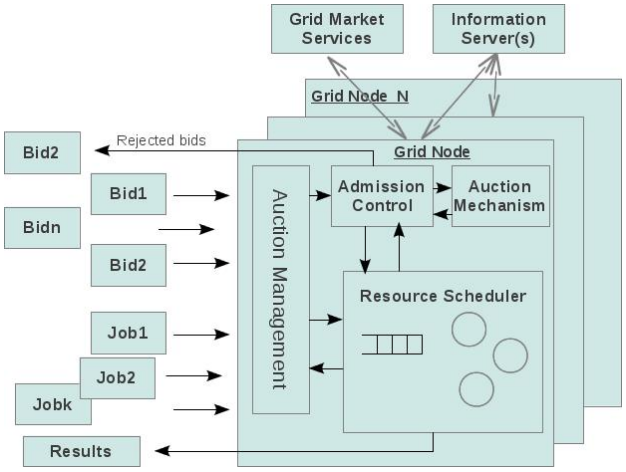


Figure 1: An auction mechanism proposed by Buyya *et al.*

An outline of the auction mechanism is illustrated in Figure 1. The steps of an auction process are as follows:

Step 1: A service provider (GSP in the figure) announces available services and invites bids from clients.
Step 2: Clients offer their bids (and they can see what other clients offer if it is an open auction).
Step 3: Step 2 goes on until no one is willing to bid a higher price or the auctioneer stops if the minimum price line is not met.
Step 4: The GSP offers the service to the one who wins.
Step 5: The consumer uses the offered resource.

To evaluate the performance of the scheme, Buyya *et al.* conducted experiments on the World-Wide Grid (WWG). From the experiments it can be observed that, the manager selects the most powerful but cheapest resources to process a given job, as cost minimization was the top priority as long as the deadline of all jobs could be met. This indicates that their economic model certainly realizes a mechanism to effectively control the trade-off among QoS parameters, deadline and the computational cost. However, this model has a flaw such that it could not fully utilize the given resources (i.e., it may waste a huge amount of resources) since it commonly occurs a situation in which the request of clients will concentrate on specific high quality resources. In addition, it could not maximize the satisfaction of the clients

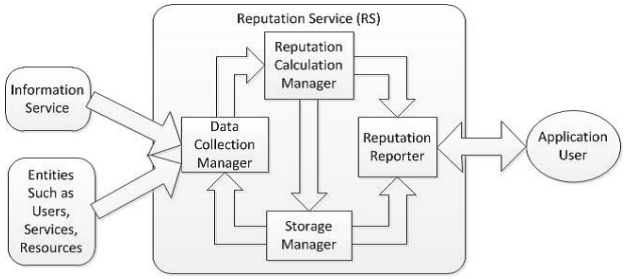


Figure 2: Architecture of a reputation service proposed by Alunkal *et al.*

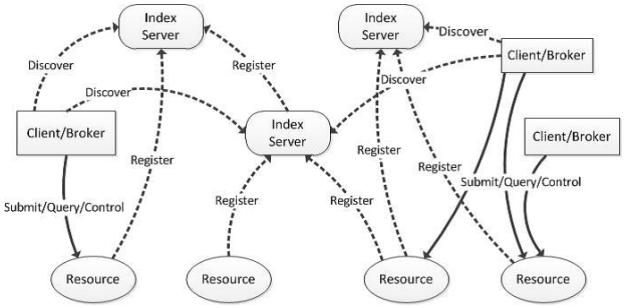


Figure 3: Interactions between resources, index servers and brokers.

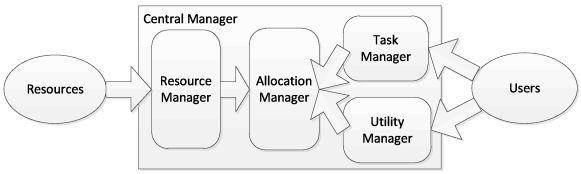


Figure 4: The structure of the central manager.

since under the auction mechanism, each client should pay much money to get a good resource although there may exist resources that are less attractive but with a reasonable price.

2.2 Reputation Model

Alunkal *et al.* introduced the notion of reputation to the resource management in Grid [1]. More concretely, their resource management scheme adopts the concepts of dynamic trust and reputation adaptation, which are based on the community experiences such as the classification, selection, and the tuning of the allocation of entities such as resources and services. They proposed a sophisticated architecture to select trusted resources which best satisfies the application requirements with respect to a predefined trust metric.

The overview of an individual reputation service is shown in Figure 2. It consists of a calculation manager, collection manager, storage manager and reputation reporter. The calculation manager is responsible for calculating the reputation value of the entities according to a prespecified context and providing the calculated values to the storage manager which stores them to maintain a global and historical view. The collection manager evaluates the quality statement, describes the requested reputation, and collects relevant data from the entities. The reporter contacts the storage manager to make a report whenever it is requested by the other clients in the Grid.

With the aid of reputation, Alunkal *et al.*'s model can reduce the influence of malicious entities, while it still ignores the satisfaction of clients. In addition, it faces to the same problem with the economic model such that the given resources could not be fully utilized in many cases.

2.3 Broker Model

Elmroth proposed a resource management scheme based on the notion of decentralized brokers [10]. The role of a broker is to select a resource which minimizes the completion time of a given job, considering the time for file staging, batch queue waiting, and the job execution. A grid architecture based on decentralized brokers is shown in Figure 3. In this architecture, each resource registers itself to at least one index server, where each index server can also register itself to index servers at a higher level; i.e., it forms a hierarchy of index servers. All clients can access resources only through their own broker, where each broker contacts index servers to discover what resources are available in the system. After acquiring the name and the contact of the resources from index servers, a broker requests individual resources for their detailed information, and performs job submission and job control by directly communicating with the resources.

3. Service Model

3.1 Model

The objective of resource management considered in this paper is described as follows. Consider a distributed system consisting of a set of resources A, a set of clients B, and a central manager C. Each resource in A models a particular service provided by the grid computer, e.g., disk space, CPU power, peripheral devices, and so on. Let $n = |A|$ denote the number of resources and $n' = |B|$ denote the number of clients. For each resource $j \in A$, the price p_j of the resource (per unit) is determined by the manager. Each client $i \in B$ is given budget e_i, and client i can request several kind of resources to maximize her profit, provided that the total expense does not exceed e_i.

The main role of central manager C is to manage the assignment of resources to the clients so that *all resources are completely assigned to the clients while maximizing the profit of each client*. In this system, such an assignment is realized by using a market model. More concretely, each

client selfishly tries to maximize its profit, and the total cost of requested resources does not exceed her budget. In order to "clear" such a market (i.e., in order to realize a situation in which all resources are assigned to clients and all clients exhaust their budget), central manager C can "control" the price of each resource. That is, by increasing the price of a resource, it becomes less attractive for all clients and it relaxes the congestion of requests to the resource, and by decreasing the price of a resource, we could attain a full utilization of the resource since it becomes attractive for all clients interested in the resource.

The objective of the central manager is to find a *market clearing price* such that a collection of selfish requests which maximizes the satisfaction of clients clears the market. In other words, under a market clearing price $\vec{p} = (p_1, \ldots, p_n)$, after each client is assigned an optimal set of resources relative to these prices, there is no surplus or deficiency of any of the resources. The structure of the central manager is illustrated in Figure 4. As shown in the figure, it consists of four components, i.e. task manager, utility manager, resource manager and the allocation manager, where the most important part is the allocation manager which is responsible for computing and allocating resources to the clients. Task manager is responsible for receiving tasks from the clients and utility manager is responsible for receiving the utility function from the clients. Finally, the role of resource manager is to manage the resources coming to or leaving from the system.

3.2 Utility Function

As was described previously, we model the satisfaction of clients by a collection of utility functions. In this paper, we assume that the utility functions are linear with respect to the amount of assigned resources. Let $x_{i,j}$ be the amount of resource j assigned to client i under price vector $\vec{p}$. Then, the utility of client i with respect to the assignment is represented by

$$\sum_{j=1}^{n} u_{i,j} \times x_{i,j}$$

using n constants $u_{i,1}, \ldots, u_{i,n}$. The reader should note that such an assumption on the linearity of utility functions makes the analysis of algorithms much easier than the case of other utility functions such as concave ones, although it is less practical than those general ones. In fact, in actual situations in the real world, it is unlikely that the degree of satisfaction "linearly" increases as increasing the amount of acquired resources, e.g., although a kid could increase her satisfaction twice as increasing the number of candies from one to two, her satisfaction does not increase to the twice by increasing the number of candies from one hundred to two hundreds.

3.3 Match Making in Grid

A typical scenario for the match making in Grid proceeds as follows. At first, all clients register their utility function to C. Tasks issued by the clients are managed by C in a synchronous manner i.e., C repeats synchronous rounds in each of which:

1) tasks issued by the clients are received at the beginning of a round,
2) tasks are assigned resources, and
3) assigned resources are used by the tasks until the beginning of the next round.

As was mentioned previously, how to calculate an appropriate assignment is the main concern in this paper.

4. Original Algorithm

4.1 Graph $N(\vec{p})$

The following price determination algorithm is borrowed from [9]. The goal of this algorithm is to find a price vector $\vec{p^*}$ such that both resources and budgets are exhausted under $\vec{p^*}$. Since the budget is limited and we are assuming linear utility functions, for any price vector $\vec{p}$, a reasonable client wishing to maximize her profit must spend her money to the resources which maximize her profit per price, i.e., $u_{i,j}/p_j$. In other words, $\alpha_i = \max_j \{u_{i,j}/p_j\}$ is considered to be the *bang per buck* for client i. Note that client i is interested in any resource j such that $u_{i,j}/p_j = \alpha_i$, and if there are several such resources, she is equally happy with any combination of such resources.

To represent such a preference of clients (under price vector $\vec{p}$), we use a directed capacitated graph $N(\vec{p}) = (V, E)$, which is defined as follows:

- $V = A \cup B \cup \{s, t\}$, where A is the set of resources, B is the set of clients, and s and t are vertices which are not contained in $A \cup B$.
- For any $j \in A$ and $i \in B$, $(i, j) \in E$, $(s, j) \in E$, and $(i, t) \in E$.
- Edge (i, j) connecting vertices in $A \cup B$ is given infinite capacity, edge (s, j) is given capacity p_j, and edge (i, t) is given capacity e_i. Note that p_j is the price of resource j and e_i is the budget given to client i.

By definition, any combination of resources allocated along the edges of $N(\vec{p})$ will make clients happiest under price vector $\vec{p}$. Computing the largest amount of resources which can be allocated in this manner, without exceeding the budget of clients or the amount of resources available, can be accomplished by computing a max-flow in $N(\vec{p})$.

With graph $N(\vec{p})$, the algorithm tries to increase the price of each resource starting from an initial price vector, by keeping an invariant such that

"$(\{s\}, A \cup B \cup \{t\})$ is a min-cut of graph $N(p)$."

See Figure 5 for illustration. This invariant ensures that all resources can be allocated to some clients (i.e., exhausted)

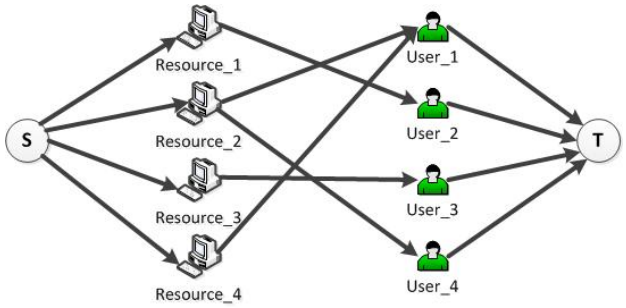


Figure 5: Network $N(p)$.

and an increase of the price vector monotonically decreases the surplus money of each client. In other words, it guarantees that when the surplus vanishes, a market clearing price vector which exhausts all resources and all budgets, is obtained.

4.2 Tight Sets

The second idea of the algorithm is to identify a set of resources whose price is to be "frozen" during the succeeding steps of the algorithm. Such a set of resources is called a tight set, which is formally defined as follows: For a set $S \subseteq A$ of resources, let $p(S)$ denote the total expense of resources in S, and for a set $T \subseteq B$ of clients, let $m(T)$ denote the total money possessed by the clients in T. Let $\Gamma(S)$ denote the set of clients adjacent to S; i.e., $\Gamma(S)$ is the set of clients who are interested in a resource in S at the current price vector $\vec{p}$. A resource set S is said to be **tight** if it holds $p(S) = m(\Gamma(S))$, i.e., if the expense of S exactly equals to the money possessed by the clients interested in S. The reader should note that clients in $\Gamma(S)$ are completely satisfied with the current prices of S, and such a satisfaction does not change even if we increase the price of the other unfrozen resources.

In the algorithm, the price of unfrozen resources monotonically increases, where instead of increasing the price of all resources at the same time, we focus on a subgraph of $N(\vec{p})$ called active graph, and try to increase the price of resources in the graph (detailed definition of the active graph will be given later). The price of resources in the active graph is conducted in such a way that it ensures that *the edges in the active graph are retained during the increasing process* (i.e., it does not become "too high"). To this end, we (uniformly) multiply the price of these resources by x and gradually increase x. It is shown in [9] that we will have a tight set when x is set to the following value:

$$x^* = min_{S \subseteq A} \frac{m(\Gamma(S))}{m(S)}.$$

Here value x^* and the corresponding tight set S^* can be found in polynomial time using n max-flow computations.

4.3 Balanced Flows

Given a flow f in network $N(\vec{p})$, the **surplus** of client i denoted by $\gamma_i(\vec{p}, f)$, is the residual capacity of edge (i, t) with respect to f, which equals to e_i minus the flow sent through (i, t). Let $\gamma(\vec{p}, f) := (\gamma_1(\vec{p}, f), \gamma_2(\vec{p}, f), \ldots, \gamma_n(\vec{p}, f))$ denote the surplus vector of clients with respect to $\vec{p}$ and f. In order to accelerate the increase of $\vec{p}$, the algorithm focuses on resources adjacent to "high-surplus" clients, since it would increase the chance of significantly increasing the price while keeping the invariant. However, the surplus of clients may take different values for different maximum flows even in the same graph. Thus, in order to well-define the surplus of clients, the algorithm defines a kind of canonical flow defined as follows: Given a price vector p, a maximum flow that minimizes $||\gamma(\vec{p}, f)||$ over all choices of f is called a **balanced flow**, where $||\vec{v}||$ denotes the l_2 norm of vector $\vec{v}$. If $||\gamma(\vec{p}, f)|| < ||\gamma(\vec{p}, f')||$, then we say f is more balanced than f', and for a given $\vec{p}$ and a flow f in $N(\vec{p})$, we denote the residual network of $N(\vec{p})$ with respect to f by $R(\vec{p}, f)$.

4.4 Main Algorithm

The main idea of the algorithm is to reduce $||\gamma(\vec{p}, f)||$ in every phase. This goal is achieved by finding a set of high-surplus clients in the balanced flow and by increasing the price of resources interested by the clients, i.e., resources adjacent to the high-surplus clients in $N(\vec{p})$. If a subset of resources becomes tight after such an increase, then we have reduced $||\gamma(\vec{p}, f)||$ since the surplus of high-surplus clients is now dropped to zero. Another possibility is that a new edge is added to $N(\vec{p})$, but such an edge will also help us to make the surplus vector more balanced.

In each phase, it identifies a subgraph of $N(\vec{p})$ induced by a set of clients $H \subset B$ and a set of resources $H' \subset A$ where initially, H is the set of clients whose surplus equals to the maximum surplus δ in B and H' is the set of resources adjacent to H. Such a subgraph is called **active graph**. As was previously described, the price of resources in the active graph increases in such a way that all edges in the graph retained, which can be ensured by multiplying the price of resources in the subgraph by x, and by gradually increasing x. Each phase is divided into several iterations. In each iteration, x increases until either a new edge is added to $N(\vec{p})$ or a subset of resources becomes tight. In the former case, we recompute the balanced flow f, and add all vertices which can reach a member of H in $R(\vec{p}, f) \setminus \{s, t\}$ to subset H. If a subset becomes tight as a result of increase, the iteration terminates. If A becomes tight, the algorithm terminates.

Finally, the initial price vector satisfying the invariant is determined as follows [9]:

- Fix the price of each resource to $1/n$.
- If there exists $j \in A$ such that $\Gamma(\{j\}) = \emptyset$ (i.e., if no client is interested in resource j due to high price), then

reduce the price of j to $\max_i\{\frac{u_{i,j}}{\alpha_i}\}$, and repeat such a modification until no such resource exists.

5. Proposed Scheme

In this section, we propose a scheme to find a market-clearing price vector in an approximated manner. The time complexity of the proposed (semi-)algorithm is mush lower than the original (exact) algorithm described in Section 4, although it merely generates an approximated solution to the problem in the sense that the amount of remaining budget is at most ϵ for any $\epsilon > 0$. Similar to the previous section, in the following explanation, we will continue to use graph $N(\vec{p})$ to represent the relation between the set of resources A and the set of clients B under the given price vector $\vec{p}$, while there are two additional vertices s and t, as before. The reader should remind that in graph $N(\vec{p})$, client i is connected with resource j by an edge if and only if j is a resource which maximizes u_{ij}/p_j.

5.1 Algorithm

Similar to the original algorithm, the proposed algorithm starts with finding a price vector which does not violate the invariant, which is ensured by making the sum of prices is less than or equal to the surplus of clients. The algorithm is divided into phases. In each phase, it identifies an active graph consisting of a set of clients $H \subset B$ and a set of resources $H' \subset A$, and increases the price of resources contained in H', i.e., the price of resources not in H' is not increased. Such an increase of the price is conducted in such a way that the edges in the active graph are retained, which is ensured by multiplying the price of all resources in H' by x and by gradually increasing x from 1, as before. Let δ be the maximum surplus in B (under current price vector $\vec{p}$). Initially, active graph is constructed such that H is a set of clients whose surplus equals to δ and H' is the set of resources adjacent to H. Let $p(H')$ be the sum of prices of resources in H' and $m(H)$ be the total money possessed by the clients in H. In order to accelerate the increase of parameter x, in the proposed algorithm, we repeat to "double" the value of x to find a sufficiently large value, and then conduct a binary search to find a value satisfying the given approximation ratio.

More concretely, in the first step, we double the value of x until one of the following conditions holds:

Case 1: $m(H) - \epsilon/n < p(H') \leq m(H)$,
Case 2: $p(H') > m(H)$, or
Case 3: (i, j) with $i \in H$ and $j \in A \setminus H'$ becomes an edge in the active graph.

In Case 1, it terminates the phase by moving all resources in H' to the frozen set. It then recomputes the balanced flow to obtain a new active graph for the next phase. If we met Case 2 for the first time, on the other hand, it starts a binary search to find x satisfying Case 1. For example, if $x = 128$, the value of x is updated as $(128 + 64)/2 = 96$. If the updated value still meets Case 2, the value further reduces to $(64 + 96)/2 = 80$ while it increases to $(128 + 96)/2 = 112$ if it meets neither of Cases 1 and 2 at $x = 96$. Such an update-and-check is repeated until it meets Case 1.

Finally, in Case 3, it tries to find the largest x using another binary search such that all resources existing in the last active graph are still being members of the new active graph, i.e., every resource in the active graph consisting of old and new resources is connected to the clients in the active graph so that the mount of budgets of those clients is at least the current price of the resource (such a condition is necessary to guarantee that all resources are exhausted). If it could not identify such x, *it terminates the algorithm without outputting a solution*. However, if there is such x, after adding new resources and all clients interested in the set of resources to the active graph, and starts the procedure from the beginning of the phase.

5.2 Analysis

The correctness and the time complexity of the algorithm are proved in the following claims.

Theorem 1: If the proposed algorithm outputs a price vector $\vec{p^*}$ as the solution, it guarantees that under $\vec{p^*}$ all resources are exhausted and the surplus of the users is at most ϵ.

Proof: By the description of the algorithm, when resources in set H' is moved to the frozen set, the surplus in H, which is the set of clients interested in H', is at most ϵ/n. Set H' is not empty even in the worst case. In addition, each user is interested in at least one resource at the initial price and all resources in A should be eventually moved to the frozen set if it outputs a solution. Thus, we can conclude that the surplus of users in B is at most ϵ after the termination of the algorithm. ■

Let M denote the total budget initially given to the clients in B, i.e., $M = m(B)$. Let k be the number of iterations spent to meet Case 1 in a phase. An upper bound on k could be given as in the following lemma.

Lemma 1: $k = O(\log \frac{nM}{\epsilon})$.

Proof: Since the doubling of x is repeated at most $\lceil \log_2 M \rceil$ times, it takes $O(\log M)$ time to identify an upper bound on the target value. After identifying the upper bound, it takes $O(\log M)$ time to identify a range of size one including the target value since the upper bound is at most M, and it needs $O(\log(n/\epsilon))$ additional time to identify the target within a range of size at most ϵ/n. Thus the lemma follows. ■

The time complexity of the proposed scheme is proved in the following theorem.

Theorem 2: The proposed scheme executes at most $O(n \log \frac{nM}{\epsilon})$ max-flow computations.

Proof: Since each phase moves at least one resource to the frozen set, the algorithm repeats the phase at most n

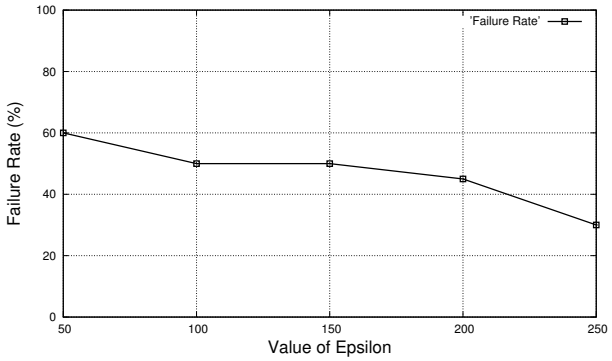


Figure 6: Failure ratio of the proposed algorithm.

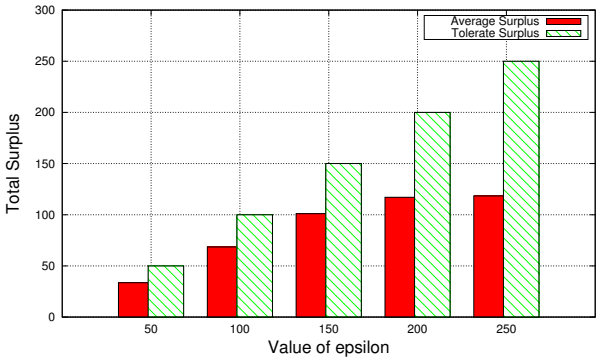


Figure 7: Guaranteed surplus and average surplus.

times. In each phase, it executes the max-flow computation in one of the following three cases: 1) at the beginning of the phase to identify an initial active graph, 2) after increasing the value of x to check which of Cases 1, 2 and 3 is met, and 3) after encountering Case 3 to identify an updated active graph. The third case occurs at most n times during the execution of the algorithm, since once a resource is included in an active graph in a phase, it is moved to the frozen set at the end of the phase or the algorithm gives up to output a solution. By Lemma 1, the increase of x is repeated $O(\log \frac{nM}{\epsilon})$ times in each phase. Thus the total number of executions of max-flow computation is given by $O(n \log \frac{nM}{\epsilon})$. Hence the theorem follows. ■

6. Experiments

6.1 Setup

We experimentally evaluated the performance of the proposed scheme using GridSim environment [5]. In the simulation, we consider a Grid consisting of 100 resources and 100 users (i.e., $n = n' = 100$). The budget of each user is 100 dollars, so that the total budget is 10000 dollars, and the initial price of each resource is randomly selected from 10 to 50 dollars. We use MIPS (Million Instructions Per Second) value to represent the characteristic of resources. The MIPS value of each resource is randomly selected from range $[300, 500]$ and the MIPS value requested by each user is randomly selected from range $[200, 400]$. The value of ϵ is varied as 50, 100, 150, 200 and 250, and for each value of ϵ, we conducted 20 runs. Finally, as the concrete max-flow algorithm, we used the Edmonds-Karp algorithm [17].

6.2 Results

The result of simulations is summarized as follows. At first, we evaluated the ratio of failed instances for each ϵ. The result is shown in Figure 6. The ratio gradually decreases as ϵ increases, and in total, it successfully outputs an approximated solution for 53% of the examined instances (even if ϵ is set to 0.5% of the given budget, it successfully outputs an approximated solution for 40% of the instances). As for the average execution time, we found that for any ϵ, it takes less than 3 sec for succeeded instances whereas it takes less than 0.5 sec for failed instances. Since the original exact algorithm proposed by Devanur *et al.* takes more than 5 min for the same instances, we can conclude that the proposed scheme effectively gives an approximated solution provided that it is executed as a preprocessor of Devanur *et al.*'s algorithm.

Parameter ϵ represents the maximal surplus at the end of succeeded computation. Thus finally, we evaluated the average surplus for the succeeded instances. Figure 7 shows the results. From the figure, we can observe that the average surplus is much less than ϵ, and specifically, we could attain an average surplus of 120 when we set ϵ to 250.

7. Concluding Remarks

In this paper, we propose a resource assignment scheme for computational grid based on the notion of market equilibrium. The proposed scheme is a semi-algorithm which outputs an approximated solution by using $O(n \log(nM/\epsilon))$ max-flow computations if it successfully terminates.

There are several issued to be addressed as the future work. The first issue is to increase the success ratio of the proposed scheme. To this end, we need to refine the scheme so that the change of the configuration of the active graph is appropriately managed. The second issue is to reduce the execution time. Although it could certainly reduce the number of max-flow computations in the original algorithm, it is still too large since the max-flow computation is an expensive task.

Acknowledgements

This work was supported in part by the Scientific Grant-in-Aid from Ministry of Education, Science, Sports and Culture of Japan and the Telecommunications Advancement Foundation.

References

[1] B. K. Alunkal, I. Veljkovic, G. Laszewski and K. Amin. "Reputation-Based Grid Resource Selection." In *Proc. of Workshop on Adaptive Grid Middleware*, 2003.

[2] D. P. Anderson and J. McLeod. "Local scheduling for volunteer computing." In *Proc. of PCGrid*, pp. 1–8, 2007.

[3] F. Azzedin and M. Maheswaran. "A trust brokering system and its application to resource management." In *Proc.of IPDPS*, p.22, 2004.

[4] L. Badia, M. Lindatrom, J. Zander and M. Zorzi. "An economic model for the radio resource management in multimedia wireless systems." *Computer Communications*, 27(11):1056–1064, 2004.

[5] R. Buyya and M. Murshed. "GridSim: A Toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing." *Concurrency and Computation: Practice and Experience*, 14(13-15):1175-1220, 2002.

[6] R. Buyya, D. Abramson, J. Giddy and H. Stockinger. "Economic models for resource management and scheduling in Grid computing." *Concurrency and Computation: Practice and Experience*, 14(13-15):1507–1542, 2002.

[7] J. Cao, D. P. Spooner, J. D. Turner and S. A. Jarvis. "Agent-based resource management for Grid computing." In *Proc. of IPCCC*, pp. 485–492, 2000.

[8] M. Caramia and S. Giordani. "Resource allocation in Grid computing: An economic model." *WSEAS Trans. Computer Research*, 3(1):19–27, 2008.

[9] N. Devanur, C. Papadimitriou, A. Saberi and V. Vazirani. "Market equilibrium via a primal-dual Algorithm for a Convex Program." *Journal of the ACM*, 55(5):1–18, 2002.

[10] E. Elmroth and J. Tordsson. "Grid resource brokering algorithms enabling advance reservations and resource selection based on performance predictions." *Future Generation Computer Systems*, 24(6):585–593, 2008.

[11] T. Friese, B. Freisleben, S. Rusitschka and A. Southall. "A framework for resource management in peer-to-peer networks." *LNCS*, Vol.2591, pp. 4–21, 2002.

[12] H. Jin. "ChinaGrid: Making Grid Computing a Reality." *LCNS*, Vol. 3334, pp.13–24, 2004.

[13] K. Miura. "Overview of Japanese science Grid project NAREGI." *Progress in Informatics*, 3:67–75, 2006.

[14] D. P. Schissel, J. R. Burruss, A. Finkelstein, S. M. Flanagan, I. T. Foster and T. W. Fredian. "Buliding the U.S. National Fusion Grid: Results from the National Fusion Collaboratory Project." *Fusion Engin. Design*, 71:245–250, 2004.

[15] H. Stockinger, F. Donno, E. Laure, S. Muzaffar and P. Kunszt. "Grid data management in action: Experience in running and supporting data management services in the EU DataGrid project." In *Proc. of CHEP*, 2003.

[16] R. Wolski, J. Brevik, J. S. Plank and T. Bryan. "Grid resource allocation and control using computational economies." In *Grid Computing: Making the Global Infrastructure a Reality*, pp.747–772, 2003.

[17] N. Zadeh. "Theoretical efficiency of the Edmonds-Karp algorithm for computing maximal flows." *Journal of the ACM*, 19(1):184–192, 1972.

Dynamic Farm Skeleton Task Allocation Through Task Mobility

Turkey Alsalkini[1], Greg Michaelson[1]
[1]Computer Science Department, Heriot-Watt University, Edinburgh, UK
(ta160, G.Michaelson) @ hw.ac.uk

Abstract—*Demand for multi-process resource invariably outstrips supply and users must often share some common provision. Where batch-based, whole processor allocation proves inflexible, user programs must compete at runtime for the same resource so the load is changeable and unpredictable. We are exploring a mechanism to balance the runtime load by moving computations between processors to optimize resource use. In this paper, we present a generic algorithmic farm skeleton which is able to move worker tasks between processors in a heterogeneous architecture at runtime guided by a simple dynamic load model. Our experiments suggest that this mechanism is able to effectively compensate for unpredictable load variations.*

Keywords: Skeleton , Mobile, Grid, Computing, Load balancing.

PDPTA'12

1 Introduction

In recent years, there has been a dramatic increase in the amount of available computing and storage, but dedicated High-Performance Computers are expensive and rare resources. Emerging multiprocessor architecture techniques offer the opportunity to integrate individual high-performance computers into a unitary high-performance system. This entails several technical challenges: difficulty of effective utilization, high communication latency, and unpredictable effective speeds.

Researchers are investigating the possibility of exploiting the computational power and resources available in global networks. Mobile computation is a way to use the resources available on both local and global networks. Mobile computation gives the programmer control over the placement of code or active computations across a network to chart and better use the available computational resources. A mobile program can transport its state and code to another location where it resumes execution [27], so in an application that uses mobile computation, the program can move between locations for better utilisation of computational resources. By using load management techniques, the program has a mechanism for distributing the tasks to worker locations to achieve performance goals (balancing the load or minimising the execution time).

The main obstacle to the commercial uptake of parallel computing is the complexity and cost of the associated software development process. A promising way to overcome the problems of parallel programming is to exploit generic programs structures, called skeleton [17]. Skeletons capture common algorithms which can be used as components for building programs. The main advantage of the skeleton approach is that all the parallelism and communication are embedded in the set of skeletons.

We are exploring a mechanism to balance the runtime load by moving computations between processors to optimize resource use. In this paper, we present a generic algorithmic farm skeleton which is able to move worker tasks between processors in a heterogeneous architecture at runtime guided by a simple dynamic load model. Our experiments suggest that this mechanism is able to effectively compensate for unpredictable load variations.

2 Background and related work

Mobility, which refers to the change of location achieved by system entities [10], involves moving computations amongst processors on a network to distribute the load, giving better use of resources and a faster performance [25, 27]. Mobility has different forms: hardware and software mobility, process migration, mobile languages, weak and strong mobility. Hardware mobility means the mobility of devices, such as laptops and PDAs. In contrast, software mobility moves the computations from one location to another location [6], typically through process migration or mobile languages. In process migration, the system determines load movement e.g. MOSIX [4], which is an operating system that supports process migration. In contrast, in mobile languages, the system gives the programmer the ability to control load movement. Weak and strong mobility are alternative forms of mobility defined by Fuggutta and Picco and Vigna [5]. Weak mobility involves moving the code from one location to another. Strong mobility involves moving the code and state information from one location to another and resuming the execution from the stop state [26]. Strong mobility is also known as transparent migration. Many mobile languages support weak and strong mobility, e.g. JavaGo [2], but Java Voyager [3] supports only weak mobility. Checkpointing is the main operation in mobile systems

to move computations amongst processors in a network or cluster by snapshotting the state of application [14], e.g. CONDOR [11].

A novel Autonomous Mobile Program (AMP) decentralised load management technique has been developed by Deng [25]. AMPs seek to execute on "better" locations and take movement decisions depending on whether the resource needs can be served locally or on another location. This movement decision also depends on future resource needs, and whether it is better to continue locally or to move to another location.

Algorithmic skeletons offer an approach in parallel programming to abstract the complexities that exist in the parallel implementations [17]. They are common parallel programming patterns that avoid the parallel and communications details for the programmer so that they are not responsible for the synchronization between the application parts. Skeletons are closely related to functional languages, so higher order functional structures can be produced by using skeletons [9]. Each skeleton has an implicit parallel implementation hidden from the application user. The main advantages of using skeletons are having a higher order programming interface and a general implementation for portability and efficiency.

Skeletons are polymorphic higher order functions, so that there are various kinds of skeletons to cover different program classes over different data types [13]. These functions are implemented by libraries. Many implementations of computations on distributed and parallel architectures support skeletal libraries which offer task parallel and data parallel skeletons. An example of a C library with MPI functions is eSkel [18], and an example of a C++ library with MPI functions is SkeTo [16].

Google developed a C++ library that offers parallel programming model, called MapReduce [12]. The MapReduce skeleton is a programming model for processing large sets of data. This model has an abstraction level where it is possible to perform computational operations while hiding communication and parallelism details, fault-tolerance, and data distribution. This model has two primitives *map* and *reduce*. The *map* operation applies a function to pairs of key/value to produce output key/value; the *reduce* operation combines the shared key results to produce the final result. The mapped function is written by the user, and the user specifies the data sets with pairs. Similarly, the reduced function is also written by the user. The closely related open-source Apache Hadoop is a Java library used to process large data sets on distributed parallel architecture such as cluster [1].

A cost or performance model may be used to estimate the costs of programs such as time and space [24, 7]. While algorithmic skeletons involve the parallelism process, communication and coordination [8], their cost models typically measure the computation and communication cost. Many cost models have been developed for algorithmic skeletons on parallel architecture. Some models determine the task placement statically [15], while others determine the whole skeleton placement dynamically [24].

Our approach is based on dynamic task placement for skeletal programming. We have developed a parallel farm skeleton using C with MPI functions which is able to move tasks between workers while preserving the execution state during moving operation. We have explored three approaches to implementing mobility in our skeleton: data mobility, data and state mobility, and data, state and code mobility.

Data mobility involves moving the data between locations on a network [14]. For state mobility, the program can correctly save its state and resume work from the saved point properly. Code mobility involves moving the whole program code, as well as the data and execution state, to a different machine [5]. This paper proposes as task mobility approach of moving the data and state for a sub-computation between processors, rather than the whole program. Our skeleton is implemented using C and MPI, but MPI clones the code to the workers so there is no need for moving the code. Code mobility is difficult to implement in heterogeneous structures, and this remains future work for our research: our work is a first step in implementing a skeleton fully able to move arbitrary code amongst machines on a network.

3 An overview of hwFarm skeleton

Our skeleton has the name *hwFarm.* In general, the main idea of skeleton is to abstract all the parallelism and communication details, but the *hwFarm* skeleton is also able to move tasks amongst the worker processors at run-time.

The hwFarm skeleton:

- is self-mobile which means that our skeleton is able to mobilize the task from one worker to another one during task execution when the overhead increases;
- supports parallelism on a distributed memory, high-performance architecture;
- hides parallelism and communication details from the program;
- presents a high-level function implemented using C and MPI [20].

3.1 Definition of hwFarm skeleton

The term task farming is used to describe parallel applications that have specific properties. Ordered and structured collections of data items, known as tasks, are each processed by the same operation. Processing the task can be performed in parallel because the tasks are independent [19]. In general, the static scheduling

of tasks to a similar number of processes gives poor load balancing. A task farm solves this by implementing dynamic scheduling to ensure a better balance. The farmer acts as the scheduler while the workers process the tasks assigned by the farmer. The h*wfarm* skeleton has the same characteristics but with the ability to move its tasks amongst workers.

The implementation is divided into three steps:

1) *Implementing skeleton with workers without mobility*: This is a simple skeleton which contains a farmer responsible for distributing the tasks to the workers executing these tasks, as shown in "Fig. 1".

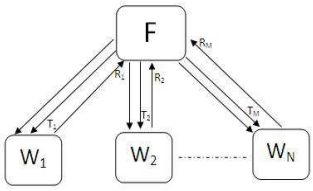


Figure 1. Standard skeleton

2) *Implementing skeleton where data has been sent between two workers*: This is the first step in making a task mobile, but the task will be processed by the worker from the beginning. See "Fig. 2".

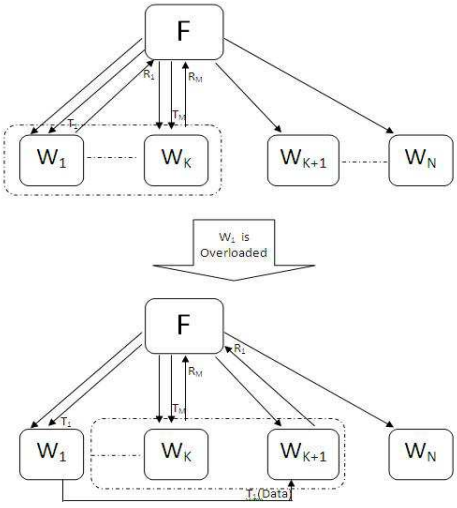


Figure 2. Skeleton with task mobility (data only)

3) *Implementing skeleton where data and state have been sent between two workers*: The skeleton can move the data and state of the task between workers. The moved elements contain the processed data and unprocessed data, so the target machine will start processing not from the beginning of the data but from the beginning of the unprocessed data. A fuller explanation of the implementation will be given in the next section. See "Fig. 3".

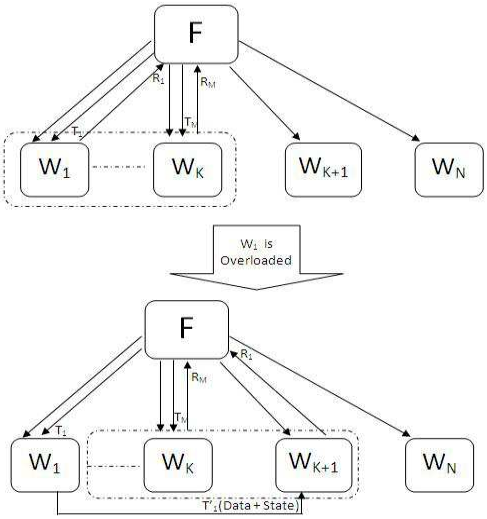


Figure 3. Skeleton with task mobility (data and state)

3.2 How to move state?

The mobility process needs to save the state of execution to continue working from the stop state. As noted, the skeleton hides the parallel and communication details from the programmer so that the programmer is not responsible for the synchronisation between application parts. The programmer has to write their own general function that should be executed by the skeleton. This function has three arguments: the input data to be processed, the output data which is the result, and parameters for the user function. An array of parameters contains the variables that the function needs to process the data. The state of execution depends on the values of these parameters. During mobility, the skeleton moves the input data that was not yet processed by the first worker, the sub-the result of the processed data and the array of parameters for the function at stop point. The new worker receives this data and continues processing from the stop point.

3.3 Movement decision:

One of the biggest issues in parallel and distributed systems is developing techniques for distributing processes to multiple locations [24, 23], to minimize the execution time and increase performance. Our skeleton balances load by using information collected from machines at run time to move a task from heavily loaded processors to lightly loaded processors.

The movable element in our skeleton is the task. The task computes the function for specific data so for task mobility we should move the function and the data. Since the function already exists in all workers, we only move the data and state between workers.

A movement decision by a skeleton depends on several polices:

- Information policy: Determines the load information to make a task placement or task mobility. This information is collected from the processors at runtime to know their load changes.
- Selection policy: Decides what task should be moved. The movement decision is taken by collaboration between the master and workers. A worker decides if its task should be moved depending on its load and the load on free workers. When the worker decides to move its task, it sends a request to the master.
- Placement policy: Identifies where a task should be transferred. The worker, after deciding that it is unable to process its task, or becomes heavily-loaded, determines the best free worker available to process the task.

3.4 Activities of hwFarm skeleton:

The sequence of activities that may happen during the execution of the skeleton is:

- The load of all workers in a system is acquired;
- The best workers are chosen where the number of workers is static. The system load in a worker, also known as its load average, is the measure of the amount of work that a computer system performs. The load average represents the average system load over a period of time. It is most easily available from a host operating system in the form of three numbers which represent the system load during the last one, five-, and fifteen-minute period [21]. The hwFarm skeleton assumes that the load for the last one-minute period;
- Each worker sends load information to the master, and then the master sends the load information to all workers;
- The tasks are distributed to the chosen workers;
- The master (farmer) awaits the results from workers and distributes new tasks;
- When the master receives a result, it will check the load for all free workers and then choose the best one to send the next task to;
- Depending on the load and percentage of increased load in this worker and the load on free workers, the worker decides if the task should be moved to a free worker or not.
- The task may then be moved from one worker to a new worker, and the destination worker continues executing the task from the stop point.

4 Experiments

In these experiments, we evaluate the performance and the behaviour of the hwFarm skeleton on heterogeneous distributed memory architecture. We use a ray tracer program that generates the image for 100 rays for 150000 objects in the scene. A ray tracing algorithm is used to produce an image by imaginary rays of light from the viewer's eye through pixels to the objects in the scene [22].

4.1 Platform:

The hwFarm skeleton is tested with a Beowulf cluster located at Heriot-Watt University. The cluster consists of 32 eight-core machines (8 quad-core Intel(R) Xeon(R) CPU E5504, running GNU/Linux at 2.00GHz with 4096 kb L2 cache and using 12GB RAM).

4.2 Evaluation:

The skeleton is tested in 4 modes:

- *Static task allocation*: The skeleton places the tasks without using load information; we will refer to this mode as *Static*. See "Table 1".
- *Dynamic task allocation*: The skeleton depends on load information collected from [processors for placing the tasks but without mobility; we will refer to this mode as *Dynamic*. See "Table 2".
- *Dynamic task allocation with load and no mobility*: The skeleton uses the load information for placing the tasks but without mobility. In this case, additional loads will be applied in different periods to some workers; we will refer to this mode as *Load*. See "Table 3".
- *Dynamic task allocation with load and mobility*: The skeleton balances the load by moving tasks from heavily loaded workers to lightly loaded workers where additional loads are applied to workers; we will refer to this mode as *Mobility*. See "Table 4".

The results of these experiments presented in the following tables:

Table 1. STATIC TASK ALLOCATION TIME(SEC)

Tasks / Workers	1	2	3	4	5
1	186.363	184.034	188.476	187.443	188.626
2	186.681	97.948	121.437	97.304	110.800
3	187.031	97.580	68.674	88.795	74.177
4	186.276	97.411	69.121	50.945	71.665
5	186.321	97.707	68.602	51.323	43.105

Table 1 shows that our skeleton gives good speed with the ray tracer program and static task allocation.

Table 2. DYNAMIC TASK ALLOCATION TIME(SEC)

Tasks / Workers	1	2	3	4	5
1	193.168	186.973	185.559	187.703	185.978
2	193.227	96.788	120.203	93.854	107.920
3	193.664	97.462	70.462	87.707	74.052
4	194.076	98.460	69.216	53.102	71.751
5	193.043	98.474	69.074	52.977	43.500

Table 2 shows show that our skeleton retains good speed with some modest difference in result from collecting and computing load information.

Table 3. DYNAMIC TASK ALLOCATION WITH LOAD AND NO MOBILITY TIME(SEC)

Tasks / Workers	1	2	3	4	5
1	335.513	292.372	316.453	309.019	294.252
2	332.121	166.745	192.063	154.189	162.842
3	297.927	167.259	113.712	140.200	121.476
4	298.318	163.596	116.965	95.658	80.906
5	300.889	151.549	110.554	93.430	79.428

Table 3 shows that the skeleton may retain speed when there is additional external load but worker performance deteriorates and execution time became slower.

Table 4. DYNAMIC TASK ALLOCATION WITH LOAD AND MOBILITY TIME(SEC)

Tasks / Workers	1	2	3	4	5
1	329.445	315.098	312.860	311.969	291.501
2	195.449	137.300	131.514	135.443	121.701
3	196.673	127.245	103.468	93.554	94.028
4	197.781	109.565	92.112	71.096	73.512
5	197.454	105.680	83.069	69.257	62.329

Table 4 shows that the performance is improved with task mobility when local loads change. However, the performance is still worse than for the *static* and *dynamic* modes.

The following graphs compare the execution time of the program with different numbers of tasks on different numbers of workers.

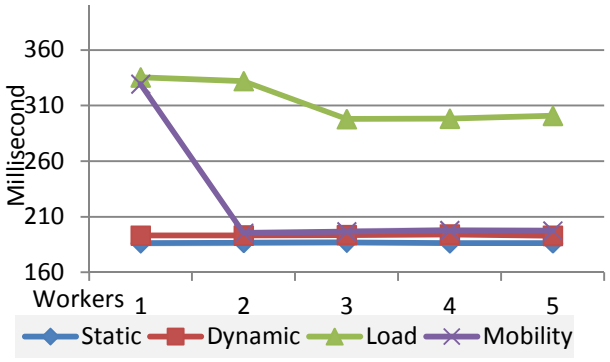


Figure 4. 1-5 Workers, 1 Tasks

Figure 4 shows the time for executing one task on 1 - 5 workers. The execution time in the *static* and *dynamic* modes is approximately the same; the difference comes from the cost of computing and collecting load. The execution time in the *load* mode depends on the load applied to the workers. In the *mobility* mode, the task is moved to a free worker when the current worker becomes unable to effectively process the task, so the execution time will be smaller. The task will not be moved when there are no free workers.

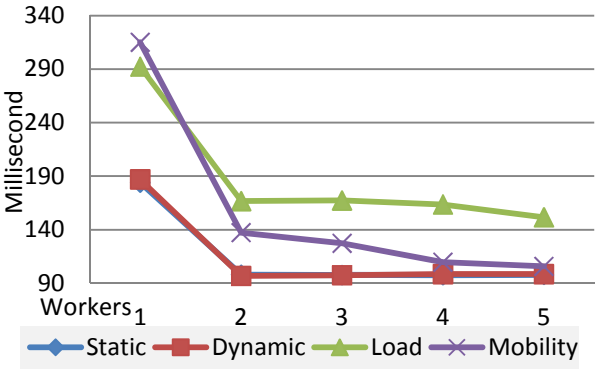


Figure 5. 1-5 Workers, 2 Tasks

Figure 5 shows the time for executing two tasks on 1 - 5 workers. In the *static* and *dynamic* modes, the execution time is approximately the same. The improvement in the *mobility* mode comes from moving the tasks from heavily loaded workers to lightly loaded workers.

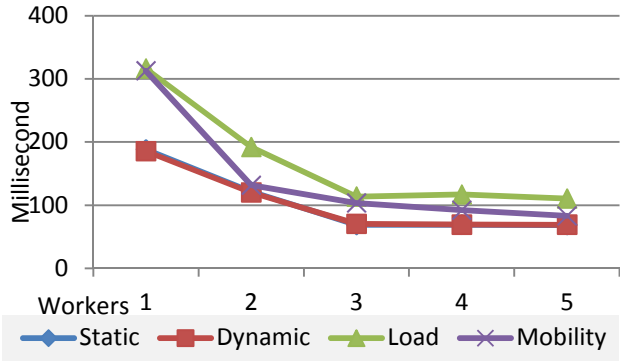


Figure 6. 1-5 Workers, 3 Task

Figure 6 shows the time for executing three tasks on 1 - 5 workers. The improvement of execution time in the *mobility* mode is related to the availability of free workers.

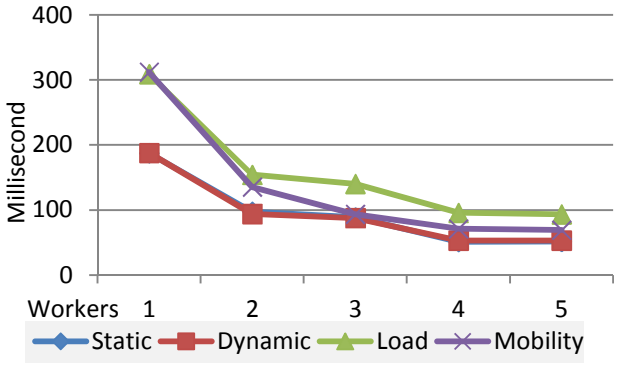


Figure 7. 1-5 Workers, 4 Tasks

Figure 7 shows the time for executing four tasks on 1 - 5 workers. The time in the *mobility* mode approaches the time in the *static* and *dynamic* modes.

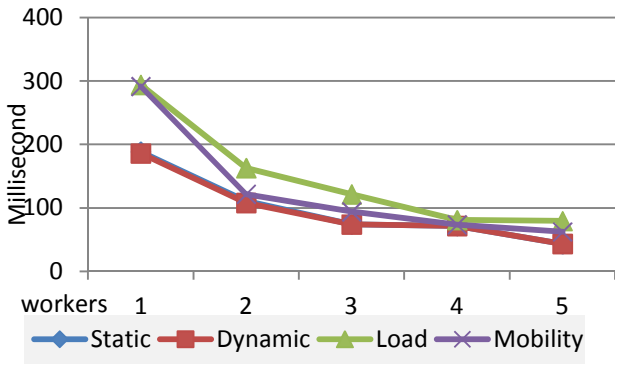


Figure 8. 1-5 Workers, 5 Tasks

Figure 8 shows the time for executing five tasks on 1 - 5 workers. The executing time in *mobility* mode is better than the executing time in *load* mode but is still worse than the *static* and *dynamic* modes.

5 Conclution and future work

We have proposed a new type of skeleton for high performance, distributed memory architecture. This skeleton is implemented using C and MPI library. This skeleton is self-mobile and able to move tasks from a heavily- loaded to a lightly-loaded worker. Our experiments show that, for the ray tracer program with small numbers of processors, the hwFarm skeleton is able to mitigate the performance effects of external load on individual processors by dynamically moving tasks across processors

We next intend to conduct considerably larger scale experiments, on much larger numbers of processors, with a variety of applications, systematically exploring mobile task behaviour in the presence of different patterns of external load. To aid this, we propose to construct a "load skeleton" which runs alongside an hwFarm application program to apply additional loads in predictable ways.

In future work, the hwFarm skeleton will be extended to be able to move code, as well as data and state, amongst processing units. In addition, we will define a richer cost model for the skeleton to take account of heterogeneity in the processing environment.

6 References

[1] Apache Hadoop Project. *http://hadoop.apache.org/*, 2010.

[2] T. Sekiguchi. "JavaGo", *http://homepage.mac.com /t.sekiguchi/javago/index.html*, May 2006.

[3] "voyager user guide," "http://www.recursionsw.com", 2005.

[4] A. Barak, S. Guday, and R. G. Wheeler, *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1993.

[5] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Trans. Softw. Eng.*, vol. 24, no. 5, pp. 342–361, May 1998.

[6] A. R. D. Bois, "Mobile Computation in a Purely Functional Language," Ph.D. thesis, School of Mathematical and Computer Science, Heriot-Watt University, United Kingdom, Aug 2005.

[7] A. Merlin and G. Hains, "A generic cost model for concurrent and data-parallel meta-computing," *Electronic Notes in Theoretical Computer Science*, vol. 128, no. 6, pp. 3 – 19, May 2005.

[8] D. K. G. Campbell, "Clumps: A Candidate Model Of Efficient, General Purpose Parallel Computation," Ph.D. thesis, Department of Computer Science, University of Exeter, United Kingdom, Oct 1994.

[9] F. A. Rabhi and S. Gorlatch, eds., Patterns and skeletons for parallel and distributed computing. London, UK: Springer-Verlag, 2003.

[10] G. Cabri, L. Leonardi, and F. Zambonelli, "Weak and strong mobility in mobile agent applications," in Proc. *2nd International Conference and Exhibition on The Practical Application of Java (PA JAVA 2000)*, Manchester (UK), April 2000.

[11] J. Basney and M. Livny, *High Performance Cluster Computing: Architectures and Systems*, Volume 1, ch. Deploying a High Throughput Computing Cluster. Prentice Hall, 1999.

[12] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51,no. 1, pp. 107–113, January 2008.

[13] J. Fischer, S. Gorlatch, and H. Bischof, Foundations of data-parallel skeletons, pp. 1–27. London, UK: Springer-Verlag, 2003.

[14] J. G. Hansen, "VIRTUAL MACHINE MOBILITY WITH SELF-MIGRATION," Ph.D. thesis, Department of Computer Science, University of Copenhagen, Apr 2009.

[15] K. Armih, Greg Michaelson, and Phil Trinder, "Cache size in a cost model for heterogeneous skeletons," In *Proc. fifth int. workshop on High-level parallel programming and applications (HLPP '11)*, 2011.

[16] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. "A library of constructive skeletons for sequential style of parallel programming," In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, New York, NY, USA, 2006. ACM. ISBN 1-59593-428-6.

[17] M. Cole, *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1989.

[18] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Comput.*, vol. 30, no. 3, pp. 389–406, 2004.

[19] M. Cole, eSkel: The Edinburgh SKEleton Library, Tutorial Introduction. Internal Paper, School of Informatics, University of Edinburgh, 2002.

[20] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference.* MIT Press, Cambridge, MA, USA, 1995. ISBN 0262691841.

[21] R. Walker, "Examining load average," *Linux J*, vol. 2006, no. 152, pp. 5--, December 2006.

[22] S. Schneider, *Concurrent and Real Time Systems: The CSP Approach (1st ed.)*. John Wiley & Sons, Inc., New York, NY, USA, 1999

[23] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Trans. Softw. Eng.*, vol. 14, no. 2, pp. 141–154, February 1988.

[24] X. Y. Deng, "Cost-Driven Autonomous Mobility," Ph.D. thesis, Heriot-Watt University, United Kingdom, May 2007.

[25] X. Y. Deng, G. Michaelson, and P. Trinder, "Cost-driven autonomous mobility," *Computer Languages. Systems and Structures*. vol. 36, no. 1, pp. 34 – 59, Apr 2010.

[26] X. Y. Deng, G. Michaelson, and P. Trinder, "Autonomous mobility skeletons," *Parallel Comput.*, vol. 32, no. 7, pp. 463–478, September 2006.

[27] Z. Kirli, "Mobile Computation with Functions," Ph.D. thesis, University of Edinburgh, Laboratory for Foundations of Computer Science:Division of Informatics, 2001.

Appendix:

Using *hwFarm* skeleton

The prototype of hwFarm skeleton is implemented in C and MPI. Our skeleton gives the programmer the ability to write their program in a sequential manner in C . They should specify the input data and identify the high-ordered function which represents our skeleton to run the program with all data in an implicit, parallel manner. Our implementation uses the MPI library to provide the communication, so we need to initialise the library before calling the skeleton.

There are some constraints on the programmer in writing the function which must have six parameters: the input data and its length, the output data and its length, and an array of parameters used in the function and its length, and these values should be initialised before function.

The main steps to write a parallel program using hwFarm skeleton are:

- Write the sequential code that should be executed in parallel on the data items as a parameterised function above.
- Initialise the MPI library.
- Initialise the input data.
- Call the hwFarm skeleton.
- Finalise the MPI library.

The prototype of the main function of hwFarm skeleton is :

```
void hwfarm(fp worker, int tasks,
        void *input,int inSize,
        int inLen, MPI_Datatype taskType,
        void* output, int outSize,
        int outLen, MPI_Datatype
        resultType, void*FunPars,
        int parsSize, int procCount)
        {...}
```

Glossary of parameters:

```
worker:     worker function.
tasks:      total number of tasks.
input:      array of input data.
inSize:     size of input data type.
inLen:      size of one task.
taskType:   type of MPI input data.
output:     array to output data.
outSize:    size of output data type
outLen:     size of data in one
resultType: type of MPI output data
FunPars:    array of function parameters
parsSize:   size of parameters
procCount:  number of processors
```

We assume that the chunk size and number of workers are static but may be made dynamic by the skeleton.

The prototype of the general function that the user writes to be called from the hwFarm skeleton is:

```
void doProcessing(
      void *inputData,int inputLen,
      void *result, int outputLen,
      void* pars, int parsSize)
      {...}
```

Glossary of parameters:

```
inputData:  input data.
inputLen:   length of input data.
result:     output data.
outputLen:  length of input data.
pars:       array of paremeters.
parsSize:   length of parameters.
```

Each worker will execute their task by calling the *doProcessing* function on their data chunk. All variables the function need should be parameterised so we can save the execution state of the function.

Scheduling Data- and Compute-intensive Applications in Hierarchical Distributed Systems

Matthias Röhm, Matthias Grabert and Franz Schweiggert
Institute of Applied Information Processing
Ulm University
Ulm, Germany
matthias.roehm@uni-ulm.de, matthias.grabert@uni-ulm.de, franz.schweiggert@uni-ulm.de

Abstract*— *The growing computerization in modern academic and industrial sectors is generating huge volumes of electronic data. Hierarchical distributed systems based on Grid and Cloud technologies promise to meet the tremendously rising resource requirements of heterogeneous, large-scale and distributed data mining applications. Scheduling plays a pivotal role in such environments. While most schedulers addressing these new challenges have a strong focus on compute-intensive applications, we introduce a new scheduling algorithm to support both compute- and data-intensive applications in dynamic, heterogeneous, hierarchical environments. The developed data-aware scheduling algorithm aims to minimize the completion times of the applications as well as their costs leading to an efficient utilization of all available resources. The algorithm is specifically designed for combined storage and compute resources as these allow jobs to be executed on resources storing the data sets and thus are the key to avoid time-consuming and expensive data transfers. Simulations and first real-world usage experiences in the Fleet Data Acquisition Miner for analyzing the data generated by the Daimler fuel cell vehicle fleet show that the algorithm is suited for the different aspects of today's data analysis challenges.

Keywords- data-intensive; scheduling; Cloud; Grid.

I. INTRODUCTION

Increasing data volumes in many industrial and academic sectors are fueling the need for novel data analysis solutions to extract valuable information. Data mining, as the key methodology to address these information needs, requires effective and efficient resource management to transform the growing data into knowledge. There have been multitudes of efforts to provide specialized resource management solutions for complex data mining scenarios, including peer-to-peer data mining, distributed data stream mining and parallel data mining [1] [2].

Recently, data mining research and development has put a focus on highly data-intensive applications. Google's publications on MapReduce [3][4] inspired many projects working on large data sets. MapReduce frameworks, like Hadoop [5], simplify the development and deployment of peta-scale data mining applications leveraging thousands of machines. MapReduce frameworks are highly scalable because the scheduler uses data location information to avoid data movement and rather send the algorithms to the data.

Other current distributed data mining research is motivated by the sharing of heterogeneous, geographic distributed, dynamic resources from multiple administrative domains to support the cooperation of different organizations [6] [7] [8]. This field is generally referred to as data mining in Grid computing environments and is highly related to the Cloud computing paradigm. Most research focused on compute-intensive applications following scheduling principles that are correct for compute-intensive, but not for data-intensive data mining applications. Though different scheduling algorithms have been proposed to optimize the relation between data transfer and execution time [9][10] [11], for data-intensive applications where the limiting factor is not CPU-power but rather storage and network speed, the underlying architecture and environment assumptions may lead to non-optimal schedules. In addition, with the almost unlimited resources available through Cloud providers the traditional scheduling concept where jobs have to be assigned to a set of limited resources is not valid any more. Instead the scheduler has to assign jobs to the resource(s) that best fits the needs of the job while considering the cost to execution time ratio.

As current data mining applications are both compute-*and* data-intensive we developed an architecture based on the notion of combined compute and storage resources to bring the advantages of the MapReduce paradigm into worldwide, heterogeneous general-purpose computing environments [12]. In this article we present a multi-objective scheduling algorithm for this dynamic, hierarchical Grid architecture incorporating the Cloud computing resource concepts. This article is organized as follows: First, we briefly introduce the generalized architecture and the implications to data- and compute-intensive scheduling. Then we describe the developed multi-objective scheduling algorithm and compare it with existing Grid scheduling algorithms. Finally, we present the simulation results of the developed scheduling algorithm.

II. A GENERALIZED GRID SCHEDULING ARCHITECTURE

Grid scheduling algorithms are responsible for mapping application resource requests to available resources. In com-

parison with other scheduling problems the scheduler has to make a decision under the following environmental characteristics:

- An application is represented by one or more jobs which might have dependencies. Each job consists of an executable with parameters, a set of input and output data specifications and requirements. The execution time for a given job is not known in general. A job may require one or more compute resources.
- A data set may be stored on multiple storage resources.
- The Grid environment is dynamic and resources are heterogeneous. There is no reliable source of information.
- Jobs arrive at an unpredictable rate.

The main goal of the Grid scheduler is to produce a schedule for all jobs arriving over time that minimizes a given objective function - makespan, average completion time or cost - under these constraints. As this scheduling problem is related to problems that are known to be NP-complete, there is little chance a polynomial algorithm exists to solve it [13]. Therefore Grid Scheduling algorithms use the structure of the Grid environment to implement heuristics or approximations.

For compute-intensive application scenarios the main resource and the limiting factor is CPU-power and the focus of the Grid scheduler is to efficiently use the compute power of multiple compute clusters. In these scenarios it is commonly assumed that the time needed to transfer the input and output data is relatively small compared to the overall execution time. These assumptions lead to the following architecture which forms the basis of current Grid schedulers:
(1) Specialized storage servers to store input and output data as well as executables. (2) A set of compute clusters from different organizations each composed of multiple compute nodes for running the algorithms. To provide a high level of transparency, these clusters are treated as one multi-CPU resource. (3) In a traditional Grid scheduling setup multiple clusters from different organisations are connected through relatively slow wide area networks whereas the network bandwidth within an organization is assumed to be infinite. A setup like this fits the needs of compute-intensive applications: To schedule a compute-intensive application requesting n computational resources, the scheduler only has to look for a cluster that has the best n free compute resources. As data transfer time is small compared to the execution time the transfer overhead is sometimes neglected.

For most data-intensive applications this assumption can not be hold. In contrary, not CPU-power but storage and network speed are the limiting factors of data-intensive applications. Data-intensive applications therefore require new scheduling strategies as the input data transfer time may well exceed execution time. Now, the scheduler should choose compute and storage resources so that the overall time or cost - depending on the scheduling objective - is minimized. Obviously, a scheduler assuming infinite bandwidth within an organization may produce non-optimal schedules.

Another aspect of traditional Grid schedulers is the assumption that there is only a very limited resource set the jobs have to be scheduled to. But with the advances in Cloud computing almost unlimited (compute-) resources are available to the scheduler.

We identified the need for some major conceptual enhancements to traditional Grid scheduling architectures to efficiently support compute- and data-intensive applications considering Cloud resources:

(1) As the amount of data increases, data can not be efficiently stored and processed from a single storage server within a cluster but has to be distributed over multiple machines. Therefore any resource may provide storage and compute capacity. This combined resource type forms the basis for scalable data-intensive applications as data can be processed directly on the storage location. To increase storage capacity and speed, computational resources of compute clusters may become combined resources by storing data on their local disks. It is important to point out that the combined resources are suitable for data- *and* compute-intensive applications as only new functionality is added.

(2) The environment is hierarchical. Resources are mainly organized within a cluster; an organization may have multiple clusters on-site or in the Cloud; and multiple organisations may want to share their resources. This not only implies an administrative hierarchy but also the network connecting the different entities has a hierarchical structure. Resources within a cluster are generally connected through a high-bandwidth low-latency interconnect (Infiniband), whereas inter-cluster or inter-organization network speed is typically much slower.

(3) Data-intensive applications are limited by two factors: the storage and network speed. Scheduling algorithms should efficiently use these resources and avoid unnecessary input data transfer through processing the data directly on the resources storing the data or resources nearby.

(4) There exist internal and external resources. There are only a limited number of internal resources while any number of external resources may be added at an additional cost.

III. A MULTI-OBJECTIVE SCHEDULING ALGORITHM

A basic algorithm trying to minimize the average runtime of all jobs (J) for combined compute and storage resources in Grids was presented in [14]. With the increasing availability of Cloud resources this basic scheduling has to be adopted to include the cost of the Cloud resources as well as the cost associated to the wide area networks needed to integrate these resources. Also internal resources might be assigned with costs to encourage a more efficient usage. In addition to low job execution costs (K_j), the users are

interested in getting their results as soon as possible so that the completion time of the jobs (C_j is another dimension of the scheduling problem. To balance between the cost and the completion time of all jobs the scheduler should minimize the following objective function:

$$F(J) = \sum_{j \in J} C_j \cdot K_j \quad (1)$$

As not all jobs are known to the scheduler in advance (offline-scheduling) but rather appear over time (online-scheduling) the minimum of $F(J)$ can only be computed retrospectively. Therefore the scheduler can only schedule a subset of the jobs at a time and can only approximate the minimum of $F(J)$.

Data-intensive jobs might be defined as a tuple $j = (p, D)$ where $D = d_1, ..., d_m$ are the data sets to be processed by program p. The task of the scheduler is to select a tuple (r, S) for each job minimizing the objective function, where r is a compute resource and $S = (s_1, ..., s_m)$ is the set of storage resources providing the m data subsets. It is assumed that all data subsets $d \in D$ have to be available on the execution resource before they can be processed by p.

The algorithm presented in this paper uses the hierarchical structure of the environment and a cost to completion time ratio to decide what set of resources should be used for a job. This dual-objective hierarchical scheduling algorithm (DOHS) is depicted in Figure 1. As described above, the algorithm has to produce a schedule with little information about the current state of the system and the jobs. Therefore the algorithm was designed to only require relative storage and compute speeds of the resources as well as the corresponding relative costs.

The inputs of the scheduling algorithm are the set of input data D, the program p, the data transfer scheduling weights α_1 to α_4 and the data to compute ratio weights β_1, β_2. As all data subsets $d \in D$ have to be available on one execution resource, the scheduler has to select a tuple (r, S) for each job where r is a compute resource and S is the set of storage resources providing the data subsets. First, the algorithm produces the set of candidate execution resources as the best (highest compute speed to cost ratio f_c) resources from each cluster in the Grid and all resources storing at least one of the data sets $d \in D$. For each of these candidate resources the set of storage resources with minimal aggregated transfer overhead f_s with regard to D is generated. From all candidates the scheduler chooses the one with the highest priority. The priority of a resource assignment (r, S, t, c) is computed as the weighted sum of the normalized transfer (t) and compute overhead (c).

The algorithm and the functions are based on the following definitions:
$P :=$ all programs available in the Grid;
$R := \{r_1, \ldots, r_n\}$ is the set of all n resources;
$D := \{d_1, ..., d_m\}$ is the m data sets of the job;
$N := \{(r, s) | r, s \in R$, r and s can exchange data directly$\}$;
$R^d := \{\ r \mid r \in R$ stores $d \in D\ \}$;
$D^r := \{\ d \mid d \in D$ is stored on $r \in R\ \}$;
c_r is the cluster of resource r;
g_r is the grid site of resource r;
sp_r is the storage speed of resources r and sc_r is its corresponding cost;
and cp_r is the compute power and cc_r is the compute cost of resource r with respect to program p, where $cp_r = 0$ and $cc_r = \infty$ if r does not fulfill all requirements of p. Different properties of a resource may be used to define the computing and storage power and cost of a resource, but at least the current usage has to be taken into account.

The data transfer overhead of a candidate resource tuple (r, S) is computed as the sum of all transfer overheads (r, s). Due to the incomplete environment information, especially missing or imprecise network information, the scheduler uses the weights α_1 to α_4 that represent the hierarchical structure. The data transfer overhead function f_s 13 assigns the weights α_1 to α_4 to the product of storage power and cost of s according to the distance between s and r: α_1 if d is stored on the resource itself $r = s$; α_2 if d is stored on a resource on the same cluster $c_r = c_s$; α_3 if d is on the same Grid instance $g_r = g_s$; and α_4 if d is on another Grid instance $g_r \neq g_s$. In case both resources are not able to exchange data directly, each resource needed to transfer the data set d from s to r is also considered.

FUNCTION $f_s(r \in R, s \in R, d \in D^+)$
  Choose shortest path $s_1, \ldots, s_k$ from s to r with
  $s_0 = s$ so that $(s_0, s_1), \ldots, (s_k, r) \in N$
  $t \leftarrow size(d) \cdot \sum_{i=0}^{k} sc_{s_i}/sp_{s_i}$
  if $r = s$ **then**
    $t \leftarrow \alpha_1 \cdot t$
  else if $c_r = c_s$ **then**
    $t \leftarrow \alpha_2 \cdot t$
  else if $g_r = g_s$ **then**
    $t \leftarrow \alpha_3 \cdot t$
  else
    $t \leftarrow \alpha_4 \cdot t$
  end if
  return t
END FUNCTION

As can easily be seen the data transfer scheduling weights α_1 to α_4 may be chosen to approximate the actual network bandwidth topology and cost or can be used minimize inter-cluster or inter-organization transfers.

FUNCTION $f_c(p \in P, r \in R)$
  if r fullfils not all requirements of p **then**
    return ∞
  end if
  $nc_r \leftarrow$ number of CPUs of r

FUNCTION DOHS ($p \in P$, R, $D \subseteq D^+$, α_{1-4}, β_{1-2})
  $R_m \leftarrow \{\, \hat{r} \mid \hat{r} \in R \wedge f_c(p,\hat{r}) = \min\{f_c(p,r) \mid r \in R \wedge c_{\hat{r}} = c_r\}\,\}$
  $R_s \leftarrow \{\, r \mid r \in R^D \wedge f_c(p,r) < \infty\,\}$
  $Z \leftarrow \emptyset$
  for all $r \in R_m \cup R_s$ **do**
    $t_r \leftarrow 0\,, \quad S_r \leftarrow \emptyset$
    for all $d \in D$ **do**
      Find $\hat{s} \in R^d$ with $f_s(r,\hat{s},d) = \min\{f_s(r,s,d) \mid s \in R^d\}$
      $t_r \leftarrow t_r + f_s(r,\,\hat{s},\,d)$
      $S_r \leftarrow S_r \cup \{\hat{s}\}$
    end for
    $Z \leftarrow Z \cup \{(\, r,\, S_r,\, t_r,\, f_c(p,r)\,)\}$
  end for
  $t_{min} \leftarrow \min\{\, t \mid (r,S,t,c) \in Z\,\}$
  $c_{min} \leftarrow \max\{\, c \mid (r,S,t,c) \in Z\,\}$
  Find $(\hat{r},\hat{S},\hat{t},\hat{c}) \in Z$ with $\beta_1 \cdot \frac{t_{min}}{\hat{t}} + \beta_2 \cdot \frac{c_{min}}{\hat{c}} = \max\{\beta_1 \cdot \frac{t_{min}}{t} + \beta_2 \cdot \frac{c_{min}}{c} \mid (r,S,t,c) \in Z\}$
  return $(\hat{r},\hat{S},\hat{t},\hat{c})$
END FUNCTION

Figure 1. DOHS algorithm for scheduling a program with multiple data sets

  $u_r \leftarrow$ currently reserved CPUs of r
  $cpu_r \leftarrow$ CPUs speed of r
  $cpu_{min} \leftarrow \min\{cpu_o \mid o \in R \wedge nc_o > u_o\}$
  if $nc_r = u_r$ **then**
    $cp_r \leftarrow cpu_r$
  else if $nc_r \leq u_r \wedge \exists o \in R$ with $nc_o > u_o$ **then**
    $cp_r \leftarrow (nc_r \cdot cpu_{min})/(nc_r + u_r)$
  else
    $cp_r \leftarrow (nc_r \cdot cpu_r)/(nc_r + u_r)$
  end if
  return cc_r/cp_r
END FUNCTION

The compute overhead function f_c 15 returns the compute power to cost ratio of the resource with respect to program p. If r does not fulfill all requirements of p the function returns ∞. In case there is at least one free CPU, the compute power is simply the CPU speed of the resource. If all CPUs are used but another resource in the Grid has a free CPU, the compute power is defined as the product of the number of CPUs times the CPU speed of the slowest available resource divided by the number of CPUs plus the reserved CPUs. Using the speed of the slowest available resource ensures, that the compute power of a busy resource is never higher then the compute power of a resource with free CPUs. If there is no free CPU in the Grid, the compute power is defined using the resource's CPU speed.

IV. Simulation results

To evaluate the presented algorithm we developed a simulation environment for executing data- and compute-intensive jobs in Grids with additional Cloud resources. In the first step of a simulation a random number of resources (> 100), clusters and organisations are created. Compute, storage and network speeds of each resource are randomly generated as well. Also the number of jobs, the number of computations per MB and the data sets per job are generated randomly. In the next step different scheduling algorithms are used to schedule the jobs as they arrive over time in the created Grid environment. For each algorithm the resulting schedule is simulated and the exact cost and time consumption of each job is computed. Based on the consumed cost and time $F(J)$ is calculated for each algorithm.

Following the Monte Carlo simulation approach 20 of these simulations were conducted and the $\sum_{j \in J} C_j \cdot K_j$ of each was recorded. As a benchmark we use a brute-force algorithm evaluating all possible resource combinations for each job minimizing different objective functions. As this requires $n \cdot s^m$ (n compute resources, jobs have in average m data sets and each data set is stored in average on s resources) objective function evaluations it is not feasible to use this algorithm for real-world scheduling. But as a benchmark, it ensures that the (locally) optimal resources are chosen according to the objective function. In contrast to the developed algorithm the benchmark algorithm is also provided with all environment and job information, including exact network bandwidth and the job's execution time. The

benchmark algorithm is used to compute a schedule based on the following objective functions that are commonly used for grid scheduling:

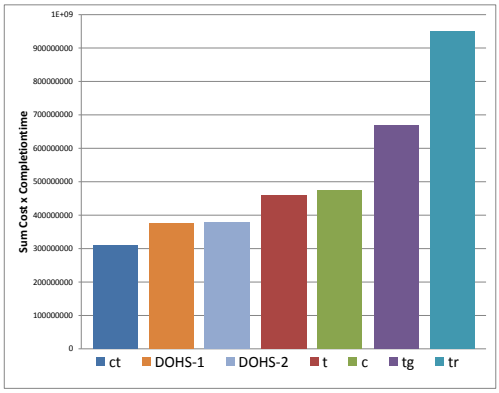
- *Cost-Time*. Minimize the product of the jobs cost and completion time.
- *Time*. Minimize the completion time of the job.
- *Cost*. Minimize the cost of the job.
- *Transfer*. The scheduler chooses an execution resource that has a minimum transfer overhead.
- *Time-Grid*. Minimize the completion time of the job based the assumption that transfer overhead and cost is zero within an organisation.

As shown in Figure 2 the developed DOHS scheduling algorithm provides good performance compared to the benchmark algorithm using the different objective functions. The DOHS algorithm achieves almost the same performance as the benchmark algorithm using the Cost-Time objective function and all information. The Time-Grid objective function may be regarded as a representative of current Grid schedulers that assume that transfer overhead and cost is zero within an organisation. It does not only require the more execution time but also consumes the more costs as the Time objective function, especially for data-intensive applications. Only the Transfer objective function, which can be seen as a generalization of the MapReduce scheduling approach of minimizing the transfer overhead, is worse. We evaluated the algorithms for a mix of compute- and data-intensive jobs. Figure 2 a) shows two different values for the DOHS algorithm. The DOHS-1 represents a scenario where users classify the jobs as compute-intensive ($\beta_1 = 0.9$, $\beta_2 = 1$) or data-intensive ($\beta_1 = 1$, $\beta_2 = 0.9$). DOHS-2 shows the results for a scenario where users do not provide any classification ($\beta_1 = 1$, $\beta_2 = 1$). As the DOHS algorithm provides parameters to adapt to different resource environments and job characteristics it was configured with the following parameters: $\alpha_1 = 1$, $\alpha_2 = 1.5$, $\alpha_3 = 5$ and $\alpha_4 = 50$.

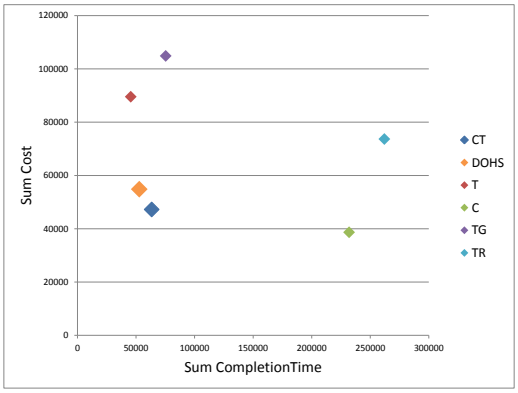
V. Related work

Recently, various systems and approaches to grid-based data mining and data-intensive scheduling have been reported in the literature. Some of those that are particularly relevant to this work are briefly reviewed here.

The GridBus resource broker [15] provides functions for scheduling data- and compute-intensive applications. In combination with the Storage Resource Broker[16] GridBus is able schedule data-intensive jobs based on various different metrics, including network bandwidth and utilization. The GridBus scheduler as most heutistic Grid schedulers, including the DIANA scheduler [9], follows the common separation between storage and compute resources, requires detailed information about the jobs and the environment and also assumes that transfer overhead within an organization is zero.



(a) $\sum_{j \in J} C_j \cdot K_j$



(b) $\sum_{j \in J} K_j$ vs $\sum_{j \in J} C_j$

Figure 2. Scheduling simulation results.

Another class of Grid schedulers uses genetic algorithms to solve the data-intensive scheduling problem [17][18]. The main disadvantages of these approaches are the computational complexity of the genetic algorithm and the requirement to have detailed information about the environment.

Hadoop [5] is the most well known open source implementation of Google's MapReduce paradigm. Hadoop's MapReduce framework is build on top of the Hadoop distributed file system (HDFS) containing all data to be mined. The map and reduce functions are typically written in Java, but also executables can be integrated via a streaming mechanism. MapReduce frameworks like Hadoop do not offer the functionality to efficiently execute compute-intensive applications on a cluster, making them unsuitable for a general-purpose data mining system. Hadoop On Demand in combination with the SUN Grid Engine try to overcome these limitations by running Hadoop on top of

a cluster management system, thus adding another layer of complexity. Still, the resources to use for MapReduce are reserved exclusively for Hadoop and can not be used by other compute-intensive jobs. Hadoop and similar MapReduce frameworks simplify the development and deployment of data-intensive applications on local clusters and cloud resources but are currently not suited for large-scale, heterogeneous environments comprised of multiple independent organizations.

Anteater [1] is a web-service-based system to handle large data sets and high computational loads. Anteater applications have to be implemented in a filter-stream structure. This processing concept and its capability to distribute fine-grained parallel task make it a highly scalable system. Due to the restriction on a filter-stream structure Anteater shares some downsides of MapReduce frameworks: Applications have to be ported to this platform which makes it almost impossible to integrate existing applications.

VI. Conclusion

In this article we introduced a multi-objective scheduling algorithm for data-intensive applications in Grid environments. The new concept of combined Grid resources in combination with the developed data location aware scheduling algorithm provides an infrastructure to build scalable data-intensive applications in worldwide, heterogeneous environments. The scheduling algorithm also supports compute-intensive applications so that a single environment can be used for both data- and compute-intensive applications. In addition the DOHS algorithm is specifically designed for Grid environments with Cloud resources where information is generally scarce. The simulation results show that the algorithm is competitive or even surpasses current Grid schedulers requiring detailed information.

Future work may focus on additional Cloud related topics such as setup time of a resource or dynamic cost.

References

[1] D. Guedes, W. Meira, and R. Ferreira, "Anteater: A service-oriented architecture for high-performance data mining," *IEEE Internet Computing*, vol. 10, no. 4, pp. 36–43, 2006.

[2] S. Datta, K. Bhaduri, C. Giannella, and H. Kargupta, "Distributed data mining in peer-to-peer networks," *IEEE Internet Computing*, vol. 10, no. 4, pp. 18–26, 2006.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," 2004, pp. 137–150. [Online]. Available: http://www.usenix.org/events/osdi04/tech/dean.html

[4] S. Ghemawat, H. Gobioff, and S. T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[5] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, 2009.

[6] V. Stankovski, M. Swain, V. Kravtsov, T. Niessen, D. Wegener, M. Röhm, J. Trnkoczy, M. May, J. Franke, A. Schuster, and W. Dubitzky, "Digging deep into the data mine with datamininggrid," *IEEE Internet Computing*, vol. 12, no. 6, pp. 69–76, 2008.

[7] A. Congiusta, D. Talia, and P. Trunfio, "Distributed data mining services leveraging wsrf," *Future Generation Computer Systems*, vol. 23, no. 1, pp. 34–41, 2007.

[8] B. Peter and W. Alexander, "Grid-aware approach to data statistics, data understanding and data preprocessing," *International Journal of High performance Computing and Networking*, vol. 1, no. 6, pp. 15–24, 2009.

[9] R. McClatchey, A. Anjum, H. Stockinger, A. Ali, I. Willers, and M. Thomas, "Data intensive and network aware (diana) grid scheduling," *Journal of Grid Computing*, vol. 5, pp. 43–64, 2007.

[10] S. Venugopal and R. Buyya, "A set coverage-based mapping heuristic for scheduling distributed data-intensive applications on global grids," in *In proceedings of the 7th IEEE/ACM International Conference on Grid Computing(Grid06*. IEEE CS press, 2006.

[11] K. Ranganathan and I. Foster, "Decoupling computation and data scheduling in distributed data-intensive applications," in *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society, 2002, pp. 352–358.

[12] M. Röhm, M. Grabert, and F. Schweiggert, "A generalized mapreduce approach for efficient mining of large data sets in the grid," in *1. International Conference on Cloud Computing, GRIDs, and Virtualization, CLOUD COMPUTING 2010*, Lisbon, Portugal, 2010, pp. 14–19.

[13] S. G. Akl and F. Dong, "Scheduling algorithms for grid computing: State of the art and open problems," School of Computing — Queen's University, Kingston, Ontario, Canada, Tech. Rep. 2006-504, January 2006. [Online]. Available: http://techreports.cs.queensu.ca/papers/view/2006-504

[14] M. Röhm, M. Grabert, and F. Schweiggert, "An integrated approach for data- and compute-intensive mining of large data sets in the grid," *International Journal On Advances in Intelligent Systems*.

[15] S. Venugopal, R. Buyya, and L. Winton, "A grid service broker for scheduling e-science applications on global data grids," *Concurrency and Computation: Practice and Experience*, vol. 18, pp. 685–699, 2006.

[16] A. Rajasekar, M. Wan, and R. Moore, "Mysrb & srb: Components of a data grid," in *HPDC*, 2002, pp. 301–310.

[17] T. Phan, K. Ranganathan, and R. Sion, "Evolving toward the perfect schedule: Co-scheduling job assignments and data replication in wide-area systems using a genetic algorithm," in *JSSPP*, 2005, pp. 173–193.

[18] A. K. M. K. A. Talukder, M. Kirley, and R. Buyya, "Multiobjective differential evolution for scheduling workflow applications on global grids," *Concurr. Comput. : Pract. Exper.*, vol. 21, no. 13, pp. 1742–1756, Sep. 2009.

A SLA-based Cloud Computing Framework: Workload and Location Aware Resource Allocation to Distributed Data Centers in a Cloud

Seokho Son, Gihun Jung, and Sung Chan Jun
School of Information and Communications,
Gwangju Institute of Science and Technology, Gwangju, Republic of Korea

Abstract - *As the number of users of cloud services increase all over the world, cloud service providers keep deploying geographically distributed data centers more. Since resource capacity of a data center is limited, cloud service providers need to distribute load to data centers. However, when the provider distributes load, SLAs (service level agreements) established with consumers should be guaranteed in an environment where a cloud provider operates geographically distributed data centers. Therefore, this paper proposes a SLA-based cloud computing framework to facilitate temporal and locational load-aware resource allocation. The contributions of this paper include 1) design of a cloud computing framework including an automated SLA negotiation mechanism and a workload and location aware resource allocation (WLARA) and 2) implementation of an agent-based cloud test bed of the proposed framework. Empirical results using the test bed show the proposed WLARA performs better than other related approaches in terms of guaranteed SLAs.*

Keywords: Cloud computing, Distributed data centers, Resource allocation, VM placement, SLA negotiation, SLA management

1 Introduction

Cloud computing is a computing paradigm that provides computing resources as services through a network to Cloud users. A cloud consists of a type of parallel or distributed systems that consists of combinations of interconnected computing resources and virtualized computing resources [1]. There are many existing cloud computing environments such as Amazon EC2 and Amazon S3 [2]. Considering such interests and business willingness of IT leaders, cloud computing paradigm is already important and to be essential in various business. In such cloud computing environments, a cloud service user consumes cloud resources as a cloud service and pay for the usage of the service. Before a cloud provider provisions a service to a consumer, the cloud provider and consumer need to establish a SLA in advance. The SLA is an agreement including the level of QoS (quality of service) between a service provider and a consumer. Usually, an SLA includes the price of a service, and the level of QoS can be adjusted by the price of the service. For instance, a cloud provider can charge a higher price to a consumer who requires a high level of QoS.

According to the increased number of cloud service users all over the world, cloud service providers keep implementing geographically distributed data centers. Since the resource capacity is limited in a data center, a cloud providers need to distribute resource load to the data centers for system performance and stability. In addition, resource load can be distributed in temporal manner since resource load in a cloud generally fluctuates by time [3]. Because of the limited resource capacity, it is hard for cloud providers to handle resource demand that exceeds the resource capacity. Therefore, to cloud providers, a load balancing scheme is a very important issue to design a cloud computing framework, and it is directly related with cloud providers' profit. Whereas it is important to design a cloud computing framework that facilitates efficiency and stability of a system, SLA-based cloud computing framework considering both temporal and locational resource load have not been sufficiently studied so far. Also we need to consider a load distribution for a cloud provider who operates geographically distributed data centers.

In [4], B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster provided a comparison between OpenNebula and several well-known virtual infrastructure managers such as Amazon EC2, vSphere, Nimbus, and so on. The comparison includes several resource placement policies (i.e., resource allocation policies) of virtual infrastructure managers. For example, there are static-greedy resource selection, round robin resource selection, and placement considering average CPU load. However, whereas the resource placement schemes are focused on placing resources in a data center, they are not focused on resource placement in which a cloud provider operates geographically distributed data centers. With geographically distributed data centers, we need to consider response time violation because of geographical distance. In addition, [5] investigated energy-aware resource provisioning and allocation algorithms that provision data center resources to client applications in a way that improves the energy efficiency of the data center. However, whereas [5] guides research directions in resource allocations, [5] does not consider a cloud provider that operates multiple

geographically distributed data centers to balance resource load and response time by geographical distance, and [5] does not include a specific negotiation mechanism.

[6] considers load placement policies on cooling and maximum data center temperatures in cloud providers that operate multiple geographically distributed data centers. Whereas [6] proposes dynamic load distribution policies that consider all electricity-related costs as well as transient cooling effects, [6] does not focus on guarantees of SLA. [6] noted that the policies delay jobs to avoid overheating or overloading data centers and violate SLAs sometimes in their simulation configuration. Moreover current providers such as Amazon EC2 do not employ sophisticated load placement policies, and the consumers themselves manually select a data center to place their virtual machines.

There are several automated negotiation mechanisms for grid or cloud resource negotiation ([7] for a survey). These negotiation mechanisms are designed for price negotiation, but these mechanisms have a lack in considering other SLA issues such as service time slot and response time. Also, they do not consider both temporal and locational load distributions to geographically distributed data centers.

Hence, this paper proposes a cloud computing framework to facilitate temporal and locational load-aware resource allocation, and we implement a test bed of the proposed cloud framework to verify the usefulness of the proposed framework in a case study. The contribution of this paper are 1) design of a cloud computing framework to facilitate temporal and locational load-aware resource allocation, and 2) implementation of a test bed of the proposed cloud computing framework. Using the proposed system, a cloud consumer can establish SLA about service price, time slot, and response time by an automated SLA negotiation scheme, besides a cloud provider can facilitate load balancing by a pricing strategy. As such, the purpose of this work is to: design a cloud computing framework facilitating resource allocation (Section 2). Also, Section 2 includes both SLA negotiation mechanism and workload and location aware resource allocation. In Section 3, we introduce the implementation of a simulation test bed for the proposed framework and evaluate performances of the proposed framework in terms of the SLA violation in Section 4. Finally, Section 5 concludes this paper with a list of future works.

2 A cloud computing framework to facilitate resource allocations

2.1 SLA-based cloud computing framework

The proposed cloud framework adopts an automated SLA negotiation mechanism to support establishing SLA. Whereas the variety of SLA options is limited for consumers within enforced SLA strategies, the different preferences between a consumer and a provider can be efficiently narrowed with an automated SLA negotiation mechanism. In case of price of a service, a negotiation mechanism between both a consumer and a provider helps to find an equilibrium satisfaction state for price. Whereas there should be many SLA issues or options in cloud services in practice, this paper focuses on 1) service price, 2) time slot to specify range of service time, and 3) service response time among several SLA issues in designing the SLA-based cloud framework. In addition to the SLA negotiation mechanism, service providers need a SLA management scheme to guarantee established SLA with a consumer. In the proposed cloud framework considers service response time to consumers as QoS. To guarantee service response time agreed with a consumer, a SLA management scheme in the proposed framework considers geographical distance between a consumer and distributed data centers of a provider when the cloud provider select a data center to allocate resources for the consumer.

In Fig. 1, the proposed framework consists of cloud service broker and cloud provider. Cloud service broker is an interface between a consumer and a cloud provider. The cloud service broker gets information of a consumer's preference for services and proceeds for cloud service discovery to find a matched service with the consumer's preference. After the broker finishes service discovery, the broker can be connected to a cloud provider who own the service. The next step is establishing SLA between the provider and the service broker on behalf of the consumer by the SLA negotiation component. If the negotiation is successful regarding service price, time slot, and response time, the broker makes the consumer pay

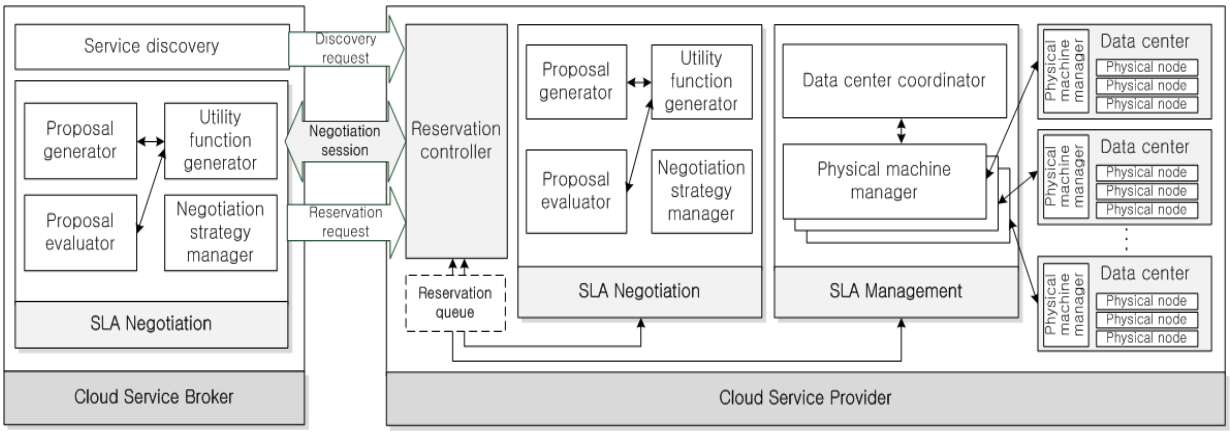


Figure 1. Design of SLA negotiation and management based cloud computing framework.

for the service in the agreed price and reserves the service.

A cloud provider consists of four components: 1) reservation controller, 2) SLA negotiation mechanism, 3) SLA management, and 4) distributed data centers. Cloud provider responds to service broker's service discovery requests. And SLA negotiation component makes a negotiation session between the provider and the broker to establish SLA. If the automated negotiation is successful, which means they came to a mutually acceptable agreement regarding price, time slot, and response time, so that an SLA has been established, the provider receives charge from the consumer through the broker.

Service reservation controller reserves the requested service by input information of the name of consumer, the service type, the start time of service, the end time of service, and the threshold of response time to the service reservation queue which is reservation list. Reservation controller sequentially checks the reservation queue to initiate the start of the services according to the start time of services. It forwards agreed response time for the services to SLA management component.

The SLA management component selects a data center among distributed data centers to allocate requested service. The conditions of selecting a data center are specified in the SLA (response time in this paper). Each data center includes a physical machine manager (PMM), and a PMM manages physical computing nodes in a data center. All PMMs monitor physical computing nodes to evaluate average response time of a data center to the consumer, and SLA management component selects a data center and specific a physical computing node by evaluated average response times. In conclusion, a service provider and a consumer can not only establish SLA about service price, time slot, and response time, but also the service provider can guarantee established SLA with a consumer through the SLA management component.

2.2 Automated SLA negotiation mechanism

In general a negotiation mechanism consists of negotiation protocol, negotiation strategy, and utility functions. Negotiation protocol is a set of rules in a communication (e.g., possible actions, language, and utterance turn) for a negotiation among negotiation parties. As a negotiation protocol, the negotiation mechanism in this work follows Rubinstein's alternating offers protocol [12] which lets agents make counter-offers to their opponents in alternate rounds. Both agents generate counter-offers and evaluate their opponent's offers until either an agreement is made or one of the agents' negotiation deadlines is reached. Counter-proposals are generated according to a negotiation strategy that consists of a concession algorithm and a tradeoff algorithm. When an agent generates a counter-proposal, the agent needs to concede a proposal since it is hard to reach an agreement without a concession. A concession algorithm determines the amount of concession for each negotiation round [13]. Also, a tradeoff algorithm is required to generate a proposal in multi-attributes negotiation. In a multi- attributes negotiation, there are multiple issues to negotiate, and a tradeoff algorithm generates a proposal by combining proposals of individual issues; the mechanism adopted a tradeoff algorithm in [14].

The utility function U(x) represents an agent's level of satisfaction of a negotiation outcome x. A negotiator (i.e., a cloud participant) can specifies utility functions. To define a price utility function, the negotiator needs to specify the most preferred price (IP, initial price) and the least preferred price (RP, reserve price). In general, the range of utility function is $\{0\} \cup [u_{\min},1]$, and U(P)= $u_{\min}$ represents the least preferred price (RP) and U(P)=1 represents the most preferred price (IP). If price is outside of IP and RP, then U(P) is 0.

To support cloud users negotiation for service price, time slot, and response time in the proposed framework, it is necessary to define utility functions of the negotiable SLA issues. In this paper, utility functions defined in [14] are adopted which are price utility function $U_P(\mathrm{P})$ and time slot utility function $U_{TS}(TS)$. Especially, the time slot utility function defined in [4] supports cloud participants to represent temporal preferences in cloud services. In addition to price and time slot issues considered in [14], this paper considers service response time as a SLA negotiation issue. The service response time represents the minimum response time that a provider provides. Let IRT (Initial response time) and RRT (Reserve response time) be the most preferred response time and the lease preferred response time respectively. A RT (Response time) given to a consumer can be evaluated by the response time utility function of a consumer $U_{RT}^{C}(RT)$ as follows,

$$U_{RT}^{C}(RT) = \begin{cases} u_{min} + (1-u_{min}) \cdot \left| \dfrac{RRT_C - RT}{RRT_C - IRT_C} \right|, & IRT_C \le RT \le RRT_C \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

Contrarily, a RT given to a provider can be evaluated by the response time utility function of a provider $U_{RT}^{P}(RT)$ as follows,

$$U_{RT}^{P}(RT) = \begin{cases} u_{\min} + (1-u_{\min}) \cdot \left| \dfrac{RT - RRT_P}{IRT_P - RRT_P} \right|, & RRT_P \le RT \le IRT_P \\ 0, & \text{otherwise.} \end{cases} \quad (2)$$

$u_{\min}$ is the minimum utility that a consumer and a provider receive for reaching a deal at their reserve prices. To differentiate between not reaching an agreement and reaching an agreement at the reserve price, $u_{\min}$ is defined as 0.01. Finally, the aggregated total utility function $U_{Total}(P,TS,RT)$ that includes price, time slot, and response time is as follows,

$$U_{Total}(P,TS,RT) = \begin{cases} 0, & \text{if either } U_P = 0,\ U_{TS} = 0,\ U_{RT} = 0 \\ w_P \cdot U_P + w_{TS} \cdot U_{TS} + w_{RT} \cdot U_{RT}, & \text{otherwise.} \end{cases} \quad (3)$$

w_P, w_{TS}, and w_{RT} are the preference weights for price, time slot, and response time, respectively. A negotiator can adjust the importance among the issues by differentiating the weights

where the weights satisfy $w_P + w_{TS} + w_{RT} = 1$. A consumer who cares price only, the consumer can assign 1 to w_P and 0 to the other weights. According to the (3), $U_{Total} = U_P$. The consumer can have a high chance to use a service in a low cost.

So, with the SLA negotiation mechanism, cloud participants can narrow preference differences (i.e., service price, time slot, and response time) so that the parties successfully reach an agreement for SLA.

2.3 Workload and location aware resource allocation

A cloud provider needs to properly allocate the provider's resources to a data center for a service provisioning so that the provider can guarantee the SLA agreed with a consumer. The response time for users depends on the utilization level of physical nodes in the data center and the location of the data center. The performance of VM depends on the allocated PM (Physical machine). If the allocated PM is under heavy workload, the performance of VM will be degraded. In addition, cloud computing services are delivered over the Internet that does not guarantee reliable delivery. Also, the response time may depend on network delay in service delivery. If the distance between a user and a data center is far, the response time of VM will be slow. So, both workload and geographical location are important factors to guarantee the SLA in terms of response time. WLARA selects a data center according to a utility function to evaluate the appropriateness in VM placement to data centers.

2.3.1 Utility function

To find an appropriate physical computing resource which fits user's SLA, this paper suggests an allocation model considering 1) the geographical location of users and provider and 2) the workload of computing resources. The proposed model uses utility function to measure an appropriateness of each data center. The utility function is described as follows:

$$U_m = \alpha \cdot U_m^{UL} + \beta \cdot U_m^{RT} \quad (4)$$

In (4), there are two terms in the utility function: 1) machine workload utility U_m^{UL}, and 2) expected response time U_m^{RT}. Each term is multiplied by the preference weight ($\alpha + \beta = 1.0$) respectively. The weight of each value can indicates that a provider's preference for placing user VM. If $\alpha \rightarrow 1.0$ and $\beta \rightarrow 0$, the provider emphasizes workload of data center in placing VM. Likewise, if $\beta \rightarrow 1.0$ and $\alpha \rightarrow 0$, the provider emphasizes finding a more close data center.

The utility function (5) for machine workload represents level of workload of a data center when the data center accepts VM placement. Let $W\tau$ be the upper bound of workload value, and W_θ and W_c be the expected workload in the data center including requested VM placement and current workload of the data center respectively. (5) shows higher utility if the placement does not affect increasing the workload of the data center. When W_θ is higher than $W\tau$ than the utility function returns the minimum utility 0.

$$U_m^{UL} = \begin{cases} 1 - \dfrac{W_\theta - W_C}{W_\tau - W_C} & , \quad W_C \le W_\theta < W_\tau \\ 0 & , \text{ otherwise.} \end{cases} \quad (5)$$

(6) describes utility function for expected response time U_m^{RT}. Let T_{SLA} be the threshold of response time in SLA, and T_e and T_c be the expected response time when a user's VM is placed in a data center and current average response time by the workload and location of the data center. This utility function returns values when T_e is in between T_c and T_{SLA}. Otherwise, the utility returns the minimum utility 0.

$$U_m^{RT} = \begin{cases} 1 - \dfrac{T_e - T_C}{T_{SLA} - T_C} & , \quad T_C \le T_e < T_{SLA} \\ 0 & , \text{ otherwise.} \end{cases} \quad (6)$$

By utility function (4), the provider can find appropriate data center for user's VM in terms of the workload and the location between user and data center.

2.3.2 Resource allocation strategy

When a provider receives the request from user, the provider asks to a reservation manager, who is in charge of managing resources to allocation and reservation, for evaluating appropriateness of each data centers. To find proper datacenter, the proposed allocation model uses utility function (4). When the reservation manager finds appropriate datacenters by the evaluation, the reservation manger asks to allocate the request of user to coordinator agent, who is in charge of managing numbers of PMs in the datacenter. Then, the coordinator receives resource allocation request, the coordinator also finds a proper PM that has light workload because we assumes that data center consists of number of PMs. Consequently, the request of user is allocated in the PM that closes to user with light workload. Therefore, when the provider uses the proposed allocation model, the request of user is allocating in the PM that guarantees reasonable response time under SLA.

3 Simulation and empical results

3.1 Simulation test bed

A simulation test bed for the proposed cloud computing framework has been implemented based on JAVA and JADE (Java Agent Development Environment) [15]. JADE is a software framework implemented by JAVA, and it is efficient to implement multi-agent systems with JADE since the implementation of JADE concretely follows FIPA (Foundation for Intelligent Physical Agents) [16] standard. The components of the proposed cloud framework were implemented as agents in the simulation test bed, and the agents communicate each other by exchanging messages needed among components by using message types defined in FIFA. The agents for the framework are as follows: 1) Service broker, 2) Service registry, 3) Reservation manager, 4) Data center coordinator, 5) PMM, and 6) PM.

3.2 Experimental settings

To show performance of the proposed framework, this experiment is focused on evaluating the number consumers who experience any SLA violation (i.e., slower response time than the response time limit descried in SLA). Also, we show 1) SLA negotiation outcomes in terms of price and response time (time slot is omitted to focus on the performance of the resource allocation in this evaluation), 2) load distribution to data centers, and 3) distance between user and data center.

Table 1 Experimant settings

SLA Negotiation Options		*Consumer*	*Provider*
SLA Preference range for price		IP:0.1($/hour) RP:1.0($/hour)	IP:1.0($/hour) RP:0.1($/hour)
SLA Preference range for response time		IRT:100(ms) RRT:250(ms)	IRT:250(ms) RRT:100(ms)
Consumer's Resources		*Range of Values*	
Number of virtual CPUs		[1, 32]	
Size of Storage		[10.0, 1000.0]	
Response time of SLA		[100, 250]	
Location		{zone 0, …, zone 9}	
Physical Machine Resources		*Values*	
Number of CPUs		{4}	
Number of Threads per CPU		{8}	
Data Center Specification		*Values*	
Number Of Physical Machines		{100}	
Storage Capacity		{1000000.0}	
Number of Monitoring Agents		{10}	
Number of Coordinator Agents		{1}	
Location		{zone 0, …, zone 9}	
Experimental Constraints		*Values*	
Allocation Model	*Reservation Manager*	{Greedy, Random, RR, IM, NIM, WLARA}	
	Coordinator and Monitoring agent	{Adaptive}	
Number of User Requests		{1000}	
Number of Data Centers		{10}	
Limitation of Workload in Physical Machines		{2.0}	

Table 1 shows input data source which is assigned to components in the test bed. Using the SLA negotiation options in Table 1, all consumers and provider select preference range for price and response time so that negotiation agents negotiate price and response time. The negotiation outcome regarding response time is then subjected to the limit of response time that the provider guarantees. At the end of each experiment, the response time of each user was calculated and checks whether the provider violated consumers' SLA or not. Also, consumer agent can have three different types of resources: 1) the number of vCPUs (virtual CPUs of a VM); 2) size of storage and 3) the VM location (Table 1). The test bed generates random value to assign types of resources as consumer's service request according to the input data range. Since the workload is depended how many virtual CPUs are waiting for allocate to physical CPU in this test bed, the number of vCPUs affects workload, and the limit of workload to a data center is bounded by 2.0. We assumed all datacenters are homogenous (i.e. same amount of resources).

In Table 1, as we described in Section 2.3, the geographical location between a data center and a consumer is an important factor that affects response time. For simulation purpose, we design ten different zones (i.e. zone 0 to 9) that represent geographical distance. In this experiment, a network delay is increasing gradually according to increasing differences between zones (20ms per each hop in the experiments). Each datacenter has a corresponding zone, and consumers can be in different zones as their locations. For each simulation, 1000 consumers, who request different amount of resources and are located in different locations, have been simulated. When a consumer agent is generated, a location is given to the consumer agent. For more realistic simulation, the distribution of locations to the ten zones follows a normal distribution (i.e. mean is 4.5, standard deviation is 2.0) so that some data centers face swamped situation with high demand while in the experiments.

The proposed framework especially the resource allocation model (i.e. Workload and Location Aware Resource Allocation, WLARA) is compared with related allocation models, which are widely used in resource allocation (i.e., VM placement) : 1) Greedy, 2) Random, 3) Round Robin (RR), and 4) Manual Zone Selection. According to the survey in [4], 1) Greedy, 2) Random, 3) RR have been used by Nimbus and Eucalyptus for placing VM to a PM in a data center. In addition, 4) the manual zone selection is a load placement scheme which is similar to current providers such as Amazon EC2. So, with the manual zone selection, consumers themselves manually select a data center to place their virtual machines. Since the manual zone selection assumes a human is aware and selects the closest data center, the manual zone selection used in this simulation makes all consumers' VM be placed to a data center deployed in the same location (i.e., zone) with each consumer. Manual zone selection is classified into ideal case and non-ideal case for a simulation purpose. With the ideal case (i.e., ideal manual selection, IM), a consumer's VM is placed to the closest data center. With the non-ideal case (i.e., non-ideal manual selection, NIM), a consumer's VM is placed to a close data center by applying random values that follow normal distribution so that we can simulate situations where users sometimes make a mistake in selecting the closest data center.

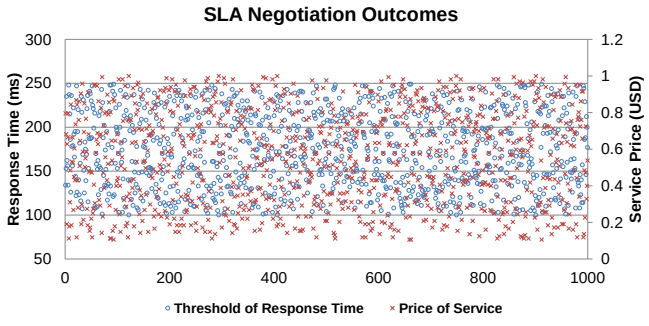
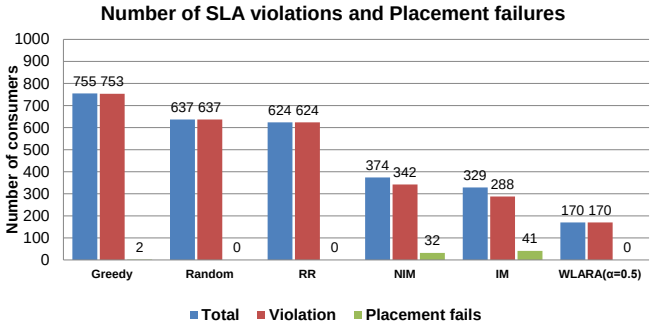
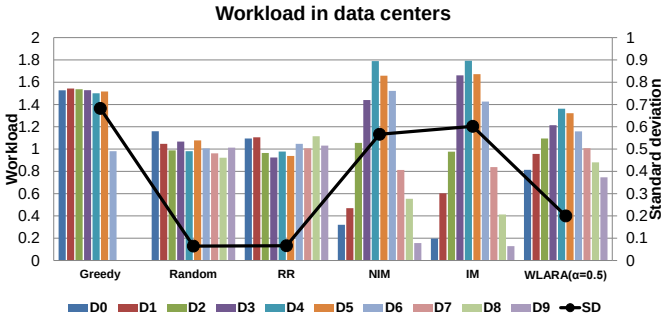


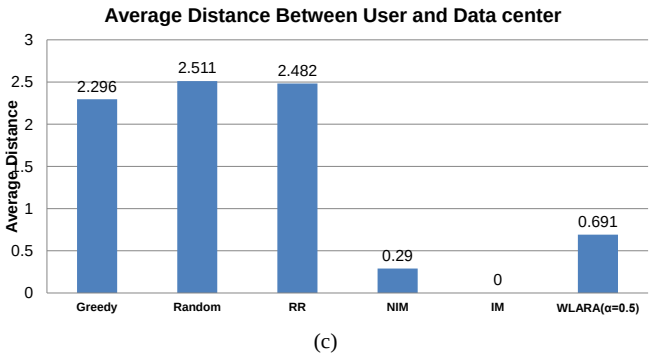
Figure 2. SLA negotiation outcomes (price and response time).



(a)

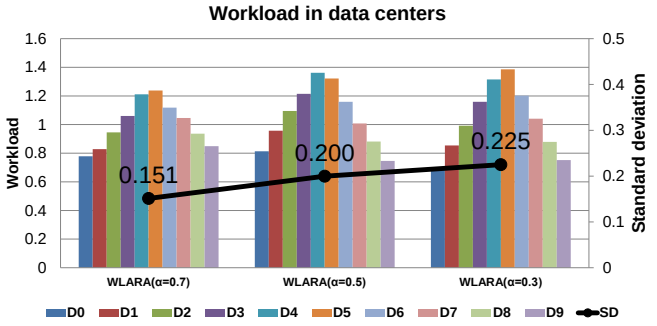


(b)

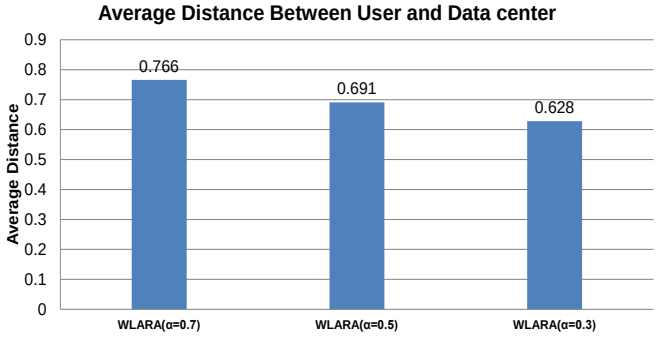


(c)

Figure 3. Performance results of resource allocation models.



(a)



(b)

Figure 4. Performance results of WLARA with varing weights ({α, β}={0.7, 0.3}, {0.5, 0.5}, {0.3, 0.7}).

3.3 Empirical results and observations

Fig. 2 shows SLA negotiation outcomes in terms of service price and response time. According to Fig. 2, SLA negotiation mechanism properly worked since threshold of response time is low (fast, respectively) when the price is low (high, respectively). Also, the response times and the prices for SLAs are well distributed (100<Agreed Response Time<250; 0.1<Agreed Price<1.0) according to the given preference range in Table 1. In addition, Fig. 3 shows the performances of all resource allocation models. According to the experiments, the proposed allocation model shows the best performance in terms of the number of SLA violations and allocation failures; 2) a balanced resource loads distribution and distance of data center. The detailed observation is described as follows.

Fig. 3 (a) shows the number of SLA violations and allocation failures. The consumers have different threshold of response time according to the outcomes of negotiated SLA. At the final stage of an experiment, the response time of each consumer's VM is aggregated to check whether it violate given response time threshold or not. In Fig. 3 (a), when the provider uses the proposed model WLARA, it guarantees the least number of SLA violations (170) and no allocation failure (0) whereas Greedy, Random, RR, NIM, and IM occurred 753, 637, 624 342, 288 SLA violations, respectively. This is because WLARA is considering both workload and response time including network delay by the utility function (4). Hence, WLARA can allocate consumer's request to a PM in a data center that has less workload and is in a close location to guarantee the threshold of response time in SLA.

Fig. 3 (b) shows load distribution in data centers. Each allocation model shows different trends of load distribution in data centers. To represent the trends, we indicated standard deviations for each model. WLARA shows less biased and smoothly distributes loads in data centers. Since the user's request is distributed to follow the random normal distribution (i.e. mean is 4.5, standard deviation is 2.0), the provider may be requested more for allocating user's VM at certain location of data center. Hence, the load distribution trend has basically in a triangle shape like WLARA and IM. The worst case is greedy model. It allocated user requests by order of data centers. Therefore, the load distribution is biased. In Fig. 3 (b), random and RR allocation model shows slight uniform distribution of load and less standard deviation. But, as shown in Fig. 3 (a), since these allocation models are not considering workload and location, the models violated SLAs too frequently.

Fig. 3 (c) shows the average geographical distance between users and allocated data centers. When the provider uses IM

selection, the average distance is zero because the IM model always places VM to the data center which has the same location with each user. IM selection is slightly better than the NIM in SLA violations. However, considering standard deviation of load distribution to data centers in Fig. 3 (b), both NIM and IM show bad performances. This may lead low performances in SLA violations and the number of allocation failures. When user request is not acceptable to selected data center due to the capacity limit, the data center may deny to allocation. The average distance with WLARA was 0 to 1. Although WLARA sometimes does not allocate user's VM to exactly same location, WLARA shows less number of SLA violations because WLARA considers both workload and response time together.

Fig. 4 represents the performance among WLARA with different experimental settings (the weights α and β in (4)). WLARA uses the utility function (4) that allows provider to adjust each weight of preference (i.e. considering workload more or location more). Hence, the results are slightly different by the weight. In Fig. 4, when the provider uses load-prefer allocation (i.e. load weight α is 0.7, and location weight β is 0.3), the distribution of load shows also best results in Fig. 4 (a). However, Fig. 4 (b) shows WLARA (α =0.7) gives relatively longer distance than with other preferences.

4 Conclusion and future work

This paper proposed a SLA-based Cloud computing framework to facilitate temporal and locational load-aware resource allocation in which a cloud provider has several data centers geographically deployed., and we implement a test bed of the proposed cloud framework to verify the usefulness of the proposed framework in a case study.

The main functionalities of the proposed cloud computing framework include a SLA negotiation mechanism and a resource allocation mechanism. Hence, by using the proposed framework, we expect cloud computing providers can facilitate distributing resource load (i.e., resource demand) in temporal and locational perspectives. In summary the contributions of this paper are as follows: 1) Design of a cloud computing framework to facilitate temporal and locational load aware resource allocation in a cloud computing environment which is similar to Amazon EC2 that has multiple data centers geographically distributed. 2) Implementation of an agent-based cloud computing test bed according to the proposed cloud computing framework. The evaluation in this paper was focused on the performance of the resource allocation algorithm. The empirical result shows WLARA performs better than other related schemes such as greedy, random, round robin, and manual selection like Amazon EC2 in terms of the number of SLA violations.

Using the proposed system, a cloud consumer can establish SLA about service price, time slot, and response time by an automated SLA negotiation scheme, and a cloud provider can facilitate load balancing by a pricing strategy. As future works, the proposed framework requires a thorough evaluation that includes cost effectiveness, resource load, average resource utilization, and so on. In addition, we need to research and classify negotiable cloud SLA issues to be included in the framework considerations.

5 Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0026438) and by PLSI supercomputing resources of Korea Institute of Science and Technology Information.

6 References

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," Future Generation Computer Systems, vol. 25, no. 6, pp.599–616, June 2009.

[2] http://aws.amazon.com/ec2/

[3] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zahara, "A view of cloud computing," Communications of the ACM, vol. 53, no. 4, pp. 50–58, 2010.

[4] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," IEEE Internet Computing, vol. 13, no. 5, pp. 14–22, 2009.

[5] R. Buyya, A. Beloglazov, and J. Abawajy, "Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges," In Proc. the 2010 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2010), pp. 6–20.

[6] K. Le, J. Zhang, J. Meng, R. Bianchini, T. D. Nguyen, and Y. Jaluria, "Reducing electricity cost through virtual machine placement in high performance computing clouds," In Proc. 2011 Super Computing (SC11), Washington, USA, November 2011.

[7] K. M. Sim, "Grid Resource Negotiation: Survey and New Directions," IEEE Trans. Syst., Man, Cybern. C, Applications and Reviews, vol. 40, no. 3, pp. 245–257, May. 2010.

[8] A. Rubinstein, "Perfect equilibrium in a bargaining model," Econometrica., vol. 50, no. 1, pp. 97–109, 1982.

[9] K. M. Sim, "Equilibria, Prudent Compromises, and the "Waiting" Game," IEEE Trans. on Systems, Man and Cybernetics, Part B: Cybernetics, vol. 35, no. 4, pp. 712–724, Aug. 2005.

[10] S. Son, K. M. Sim, "A Price- and-Time-Slot-Negotiation Mechanism for Cloud Service Reservations," IEEE Trans. on Systems, Man and Cybernetics, Part B: Cybernetics, vol. 42, no. 3, pp. 713–728, June 2012.

[11] http://jade.tilab.com/

[12] http://www.fipa.org/

Effective Heuristic Algorithm for Scheduling Workflow on Utility Grids

Vahid Khajehvand[1], **Hossein Pedram**[2], **and Mostafa Zandieh**[3]
[1]Department of Computer Engineering and Information Technology, Qazvin Branch, Islamic Azad University, Qazvin, Iran
[2]Department of Computer Engineering and Information Technology, Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran
[3]Department of Industrial Management, Shahid Beheshti University, G.C., Tehran, Iran

Abstract—*An effective workflow application scheduling on shared resources largely contributes to achieving a high performance in Utility Grids. Users share resources and these resources are autonomously managed in these environments. Users submit application-tasks to the resources, in turn, the resources eventually schedule the application-tasks due to their own allocation policy. Obviously, there is no explicit control on allocating resources to application-tasks on the part of users, the fact that results make users fail in optimizing the application makespan and allocation cost. The application scheduling problem is, in general, proven to be NP-complete. In the current paper, firstly, a Workflow Scheduling Cost-based (WSC) model is developed in order to effectively schedule an application in Utility Grids so that the application makespan and allocation-cost can be minimized. Secondly, a Minimum First-fit Cost-Makespan Trade-off (MinFCMT) heuristic algorithm is introduced to solve the WSC model. Based on the existing application scheduling algorithm, widespread simulation of the real parallel workload models as well as the synthetic workflow is exploited to evaluate the MinFCMT heuristic algorithm. The results show that the MinFCMT algorithm is more effective than the present algorithms due to optimizing the application makespan and allocation-cost in a very low runtime.*

Keywords*:* Utility Grids; Performance Optimization; Resource Provisioning; Workflow Scheduling.

1 Introduction

Grid computing can control many heterogeneous distributed resources for executing the computations and data intensive applications. Grid computing has recently been oriented towards pay-as-you-go models. In these models, the resource providers receive fees from the users for presenting computing and data services. That is why, the industry pioneers such as IBM, HP, Intel and SUN which have a large share in this business are more inclined toward the grid computing. IBM, for instance exploits "e-business on demand" model, HP exploits "Adaptive enterprise" model and Sun Microsystems apply to "pay-as-you-go" model [1].

The shared distributed infrastructures supply the grid environment software and hardware resources, in order to conduct large-scale computations. These infrastructures turned out to be efficient for executing applications in sciences such as astronomy [2], high energy physics [3], earthquake [4]. The challenge faced by the scientists in these fields is how to use cyber-infrastructure for transferring knowledge from the scientific environments to the distributed computing environments. The workflow is the most common approach to describe an application in a high level form regardless of the distributed computing environment. The workflow is represented in a "Direct Acyclic Graph" (DAG) with nodes and edges representing the tasks and data dependencies between the tasks, respectively. Once an application is transformed into the workflow structure, a workflow management system will be ready to control and manage the execution of workflow on the distributed infrastructure. In these environments, indeed, access to the shared computational resources is carried out through the queue-based Local Resource Management (LRM) system including Portable Batch System (PBS) [5] and Local Sharing Facility (LSF) [6].

In an interactive Grid computing environment the users are expecting to receive services whereas the resource providers are ready to offer services to them. The resource providers advertise the available resources planned by the application-level schedulers who receive fees upon providing services. An environment characterized with the above-mentioned users and service providers is known as Utility Grids. A competition develops among users caused by the resources-pricing policies so that users begin being involved in a competition with one another only to gain a resource with an affordable cost and an efficient processing capability. Hence, the resource providers are driven into a competition with one another to sell their idle resources to the users for more profits as well as to enhance the resource utilization. The scheduling problem becomes highly complicated and NP-complete [7] in such an environment due to the different resource consumers and providers so that each side pursues its own profits. It is worth noting that the resource consumers and providers are acting independently with conflicting aims. The resource consumers seek the minimum makespan and allocation-cost

for scheduling application, whereas the resource providers seek the resource utilization gains. Thus, the main challenge confronted by the users in this environment, will be scheduling an application on the heterogeneous resources in which the users have no explicit control so that both makespan and allocation-cost can be minimized.

The present paper deals with developing a Workflow Scheduling Cost-based (WSC) model in order to effectively schedule an application in the Utility Grids so that the application makespan and allocation-cost can be minimized. In fact, the WSC model allows the users to make a trade-off between an application makespan and allocation-cost. Next, a Minimum First-fit Cost-Makespan Trade-off (MinFCMT) heuristic algorithm is employed to solve the WSC model. The MinFCMT is a heuristic algorithm scheduling an application in a form that both makespan and allocation-cost can be optimized due to a trade-off factor. The trade-off factor shows the preference of the allocation-cost optimization to the turnaround-time. Finally, to study and evaluate the efficiency of the proposed algorithm on the proposed model, a handful of experiments have been conducted and simulated. The simulation results show that the MinFCMT algorithm is very effective in a workflow-application scheduling. The main contributions of the present paper are as follows:

- Developing a WSC model based on provisioning the resources for the workflow scheduling so that the task-execution dependencies are satisfied and its application makespan and allocation-cost can be minimized.
- Developing a MinFCMT heuristic algorithm based on the WSC model so that it is an algorithm scheduling an application in a form that both makespan and allocation-cost can be optimized due to the trade-off factor.

The rest of this paper is organized as follows: Section 2 introduces an execution environment. A proposed detailed model and heuristic algorithm is described in section 3. Section 4 involves a simulation setup and an analysis of the results and its relevant experiments in order to evaluate the efficiency of the proposed algorithm. Section 5 discusses the related works. Finally, section 6 ends with a conclusion and future work.

2 Execution environment

The agreement-based resource management allows an application-level scheduler to attain the resources in the desired time. The workflow management system, therefore, ensures access to the desired resources within the agreed time and cost. In the most resources, an agreed structure is reached between the provider and consumer in terms of available time slots. In clusters, for instance a slot indicates the availability of a number of the related processors, start time, duration and cost. Once a slot is obtained, it can subsequently be used without an extra interaction between the provider and consumer.

A workflow-application is represented in a DAG. A DAG is defined as $G = (V, E)$, where V is a set of nodes, each node representing a task, and E is a set of links, each link representing the execution precedence order between two tasks. For example, a link $(i, j) \in E$ represents the precedence constraint that task v_i needs to be completed before task v_j starts. The data is a $V*V$ matrix of the communication data, where d_{ij} is the amount of the data required to be transmitted from the task v_i to the task v_j. As a workflow may consist of sub-workflows with multiple entries and exits so the first thing to be done is to add two pseudo-tasks, a top task and a bottom task, with zero execution time indicated by 0 and $n + 1$, respectively. The top task spawns all actual entry tasks of the workflow to be linked to a single node, while the bottom task joins all actual exit tasks to a single node.

The user submits the application characteristics to the application-level scheduler only to be executed on the grid environment. The user expects to have his application executed with the minimal makespan and allocation-cost. Certainly, the users exploit trade-off factor in order to show a preference for cost to makespan. In cases where this factor is not specified by the users, the default trade-off factor is considered as equal.

In fact, the application-level scheduler acts as a mediator between the resource providers and users. Due to the reports of the available slots obtained from the resource providers, the application-level scheduler plans the application. The entire slots exploited in planning the application, will be submitted to LRM in order to provision the resources. All of the computational resources can act as a service-provider (site) for time-slots.

We suppose that the application-task performance models are clear on each resource. The execution time of a certain task, therefore, may be obtained from a certain resource due to application performance models. Also, the execution of a single task consists of three phases: (a) the input data retrieval from the resource executing the immediate predecessors of the task (b) the task execution and (c) the output data communication from the current resources to the resources presumed to execute successors of the task.

To transfer the data between the application-tasks, three data-management strategies have been proposed by Deelman et al. [8] known as the regular, dynamic cleanup and the remote I/O (on demand). In this paper, the remote I/O (on demand) strategy has been used, so that the output data are submitted to the resource that is seeking to execute immediate the successor-tasks from the immediate predecessor-tasks using the existing high-speed network among the resources.

3 The proposed model and algorithm

In general, the users are in need of two QoS: the deadline and budget of their applications on the pay-per-use services [9]. The users normally tend to run their applications in as the lowest makespan and cost as possible. Thus, a trade-off factor indicating the significance of the cost to time will be used. In this section, the issue of application scheduling will be stated and the WSC model will be presented and then solved in order to optimize the application cost-makespan. Finally, a heuristic algorithm will be developed to conduct the application scheduling with the aim of optimizing the cost and makespan.

3.1 The proposed cost-based model

The model consists of a set of heterogeneous consumers and resource providers where the consumers seek to schedule their workflow applications with the minimum cost and makespan. In the WSC model, R stands for a set of available heterogeneous resources. Each resource consists of a set of slots.

The application model is defined in section 2. If task v_i is executed on the slots available to resource r_j, then, the Task Execution Time (TET) will be TET_{ij} and the Task Allocation Cost (TAC) is obtained by

$$TAC_{ij} = TET_{ij} \times C_j \times RNP_i, \ \forall v_i \in V, r_j \in R, \quad (1)$$

where C_j is the allocation cost of r_j resource slots and RNP_i is the Required Number of Processors (RNP) of the task v_i. To make sure that each task is executed only on the slots of the same resource, binary decision variable x_{ij} is to be defined by

$$x_{ij} = \begin{cases} 1 & \text{if the resource slot of } r_j \text{ is selected for executing task } v_i \\ 0 & \text{otherwise} \end{cases}, \quad (2)$$

to achieve that, it is essential to satisfy

$$\sum_{\forall r_j \in R} x_{ij} = 1, \ \forall v_i \in V. \quad (3)$$

Now, the execution time and allocation-cost of each task v_i can be obtained, provided that the task is executed only on the slots of the same resource according to

$$TET_i = \sum_{\forall r_j \in R} TET_{ij} x_{ij}, \ \forall v_i \in V, \quad (4)$$

$$TAC_i = \sum_{\forall r_j \in R} TAC_{ij} x_{ij}, \ \forall v_i \in V. \quad (5)$$

The data transfer rates of the resources are stored in the matrix B and d_{ij} is the amount of the data supposed to be transmitted from the task v_i to the task v_j. L stands for the average latency cost of the resources. The Data Communication Cost (DCC) from the task v_i (scheduled onto the resource r_m) to the task v_j (scheduled onto the resource r_n) is defined by

$$DCC_{ij} = L + \frac{d_{ij}}{B_{mn}}, \ \forall v_i, v_j \in V, \forall r_m, r_n \in R. \quad (6)$$

The Total Data Communication Cost (TDCC) between the tasks v_i and v_j is obtained by

$$TDCC_{ij} = DCC_{ij} \left(1 - \sum_{\forall r_k \in R} x_{ik} x_{jk}\right), \ \forall v_i, v_j \in V, \quad (7)$$

in case, both tasks v_i and v_j are scheduled on the slots of the same resource, the acquired summation in (7) becomes one, as a result $TDCC_{ij}$ will be zeroed.

The Earliest Start Time (EST) of the task v_i on the resource r_j is known as EST_{ij}. The EST of the entry task is to be initialized with the Simulation Current Time (SCT) by

$$EST_{0j} = SCT, \ \forall r_j \in R. \quad (8)$$

For the rest of the workflow tasks, the EST value is computed by

$$EST_{ij} = \max_{\forall r_j \in R} \left\{ AV_j, \max_{\forall v_m \in pred(v_i)} \{SFT_m + TDCC_{mi}\} \right\}, \quad (9)$$

so that their computations recursively start with the entry task, where AV_j is the earliest start time of the slots of the resource r_j which is capable of executing the task v_i. The secondary max block in (9) computes the starting ready time of the execution of task v_i. To obtain a task ready time, firstly, execution time of the entire immediate predecessor tasks is computed. Secondly, all required data of the task v_i need to have reached the resource r_j. The Earliest Finish Time (EFT) of the task v_i on the resource r_j is known as EFT_{ij} which is computed by

$$EFT_{ij} = TET_i + EST_{ij}, \ \forall v_i \in V, \forall r_j \in R, \quad (10)$$

to compute the EFT of the task v_i, the entire immediate predecessor tasks need to be scheduled beforehand. After a task v_i is selected to be scheduled on the resource r_j (due to (11)) the EFT of the task v_i on the resource r_j is equal to the Selected Finish Time (SFT) of the task v_i.

Supposing α as a trade-off factor indicating the preference of the allocation-cost to the execution time, the trade-off cost metric of the execution of the task v_i on the resource r_j is obtained by

$$\varphi_{ij} = \alpha NTAC_{ij} + (1-\alpha) NEFT_{ij}, \ \forall v_i \in V, \forall r_j \in R, \quad (11)$$

where $NTAC_{ij}$ and $NEFT_{ij}$ are the normalized TAC_{ij} and EFT_{ij}, respectively, so that they are computed by

$$NTAC_{ij} = \frac{TAC_{ij} - \min_{\forall r_j \in R}\{TAC_{ij}\}}{\max_{\forall r_j \in R}\{TAC_{ij}\} - \min_{\forall r_j \in R}\{TAC_{ij}\}}, \quad (12)$$

$$NEFT_{ij} = \frac{EFT_{ij} - \min_{\forall r_j \in R}\{EFT_{ij}\}}{\max_{\forall r_j \in R}\{EFT_{ij}\} - \min_{\forall r_j \in R}\{EFT_{ij}\}}, \quad (13)$$

respectively. The normalization is important because we do not know what the range of values the allocation-cost and finish time will take, are for a given solution.

Thus, the objective function of the application scheduling problem is obtained by the minimization of the sum of the trade-off cost metrics for the whole application-tasks reached by

$$\min\left(\sum_{\forall v_i \in V} \min_{\forall r_j \in R} \varphi_{ij}\right). \quad (14)$$

The application scheduling problem involves mapping each task v_i onto the suitable resource r_j, so that the application makespan and allocation-cost can be minimized.

Upon the completion of the whole application tasks, makespan and allocation-cost will be computed by

$$makespan = \sum_{\forall r_j \in R} EFT_{(n+1)j} x_{(n+1)j} - \sum_{\forall r_j \in R} EST_{0j} x_{0j}, \quad (15)$$

$$allocation\ \cos t = \sum_{\forall v_i \in V} \sum_{\forall r_j \in R} TAC_{ij} x_{ij}. \quad (16)$$

In the following section, a heuristic algorithm is presented to solve the WSC model as a whole.

3.2 The proposed MinFCMT heuristic algorithm

The MinFCMT is a heuristic algorithm which is employed to solve the WSC model. This algorithm schedules an application in a form that both makespan and allocation-cost can be optimized due to the trade-off factor.

There is a handful of choices for each task, among which the choice capable of minimizing the cost metric of (11) will be selected as the best solution. According to the best solution, the *EST* needs to be computed to execute immediate successor tasks and this procedure will be carried on so long as the execution of the whole application tasks will be finished.

The MinFCMT pseudo-code is presented in algorithm 1 which operates according to the WSC model. The algorithm obtains the available slot lists to all resources and the unscheduled tasks as an input parameter (lines 1, 2). Moreover, the *EST* is initialized with simulation current time (line 3). The application-level scheduler carries out the planning of each application task due to available slots list characteristics with an eye on the cost metric presented in (11), (lines 4 to 14). Initially, a list of unplanned tasks which are eligible to be executed is selected (line 5). Next, the eligible tasks are defined as the ones whose parents' tasks execution is completed, though the very same tasks have not been executed yet. The available slots list of each resource is obtained by line 7. In line 8, the *EST* of the task *T* is computed recursively due to (9). Eventually, the *EFT* of the task *T* is computed by (10), (line 9).

The *EST* is computed on the basis of the completion-time of the latest parents tasks *T*. Next, the best slot capable of executing the task is selected for each task *T* on each resource. In cases, the selected resource does not match with the resource which executes the parents' tasks, the data-transfer time needs to be added to the *EST* due to (7) and (9).

Once the best slot for executing task *T* is obtained on each resource, the resource which minimizes the cost metric in (11) will be selected as the best resource (line 10). Now, it comes to allocating the task *T* to a selected resource (line 11) as well as updating the slots list of the selected resource (line 12). This procedure needs to be continued as long as there still exists an eligible task (lines 4 to 14). Finally, when the entire application tasks are planned, the makespan and allocation-cost need to be computed according to (15) and (16), respectively. At the end of the completion of the whole application tasks, the slots assigned to the application tasks will be released.

4 The simulation setup and the results

To conduct an experimental evaluation of the efficiency of algorithm 1, the GridSim [10] is used to simulate the application-level scheduler in the Utility Grids environment. The Grids environment which is modeled in this simulation consists of ten sites belonging to a subset of the European Data Grid (EDG) spread across five countries which are interconnected via a high-speed network [1, 11]. The workload simulated on these sites follows the workload model generated by Lublin [12]. The main purpose of the use of this model is to create a realistic simulation environment where the tasks compete with one another. Table 1 shows the workload parameters values applied to in the Lublin model. Table 2 shows the configuration of the resources on the Grids test-bed. The configuration of the resources is used in order to show the heterogeneity of the execution environment.

Table 1: Lublin workload model parameter values.

Workload parameter	Value
JobType	Batch JOBS
Maximum number of CPUs required by a job(p)	1000 $Log_2(p)$
uHi	uHi-2.5
uMed	As created by Lublin
Other parameters	model

Algorithm 1: The pseudo-code for the MinFCMT heuristic algorithm

Input: An application characteristics with an instruction length for each task and the required CPUs
The resource characteristics and the available slots to each resource

Output: The workflow scheduling

```
1   Get the list of the available time slots for all resources.
2   UnScheduledTask = get the list of the tasks which have not
    been scheduled yet.
3   Assign the simulation current time to the EST.
4   While UnScheduledTask is not empty do
5       EligibleTasks = select all tasks which executions of their
        parents have been completed.
6       for each T in the EligibleTasks do
7           Acquire the available slots of each resource.
8           Compute the EST of the task T on each resource by
            (9).
9           Compute the EFT of the task T due to (10).
10          Find a time slot (TS) which is feasible for the task T
            while minimizing the cost metrics as defined in (11).
11          Allocate the TS on the resource r to the task T
12          Update the list of available slots to the resource r.
13      end for
14  end while
15  Compute the makespan and allocation-cost of the application
    respectively, as defined in (15) and (16).
```

Table 2: Simulated EDG testbed resources.

Site name (Location)	Number of CPUs	Single CPU rating(MIPS)	Processing cost(G$)
RAL(UK)	20	1140	0.0061
Imperial College(UK)	26	1330	0.1799
NorduGrid(Norway)	265	1176	0.0627
NIKHEF(Netherlands)	54	1166	0.0353
Lyon(France)	60	1320	0.1424
Milano(Italy)	135	1000	0.0024
Torina(Italy)	200	1330	1.856
Catania(Italy)	252	1200	0.1267
Padova(Italy)	65	1000	0.0032
Bologna(Italy)	100	1140	0.0069

To conduct experiments, a parameterized graph generator is used to create a synthetic workflow application [13]. The application characteristics contain n=100 tasks with an average execution time of 1000 *s* [14]. The workflow on the average consists of $\sqrt{n}$ levels (the workflow graph depths) and $\sqrt{n}$ tasks at each level. Each task on the average needs 25 CPUs for executing. The mean value of the data transfer among the tasks is 1000 *Gb*. The mean bandwidth value among resources is 10 *Gb/s* with a mean latency time of 150 *s*.

At this stage, the scheduling algorithm that uses the best-effort QoS for scheduling, is simulated and tagged as the BE. In BE, the exploited heuristic method selects a resource with the minimum number of tasks in the waiting and running queues.

An application scheduling algorithm that uses a cost model, is presented by Singh et al. [14, 15]. This cost-modeled algorithm makes a trade-off between scheduling and allocation-cost based on trade-off factor. The scheduling takes place using a multi-objective genetic algorithm, as well as simulating the algorithm. It is tagged as the MOGA for brevity [14, 15].

The MinFCMT, the MOGA and the BE algorithms are simulated and their performance is evaluated through conducting a number of experiments. Finally, the results from the algorithms are compared with one another where *"trade-off factor=0.5"*. Due to the considerable discrepancy among the experiments' results, the scales of the y-axis of the diagram are shown in the logarithmic scales which are based on 10.

Fig. 1 reveals the simulation experiment results from the allocation-cost of the three algorithms. The MinFCMT algorithm allocation-cost declines nearly 40% compared to the MOGA algorithm as well as almost one order of magnitude compared to the BE algorithm.

Fig. 2 reveals the simulation experiment results from the makespan of the three algorithms. The makespan of the MinFCMT algorithm declines nearly 14% compared to the MOGA algorithm and almost one order of magnitude compared to the BE algorithm.

To obtain the slots for planning, the MOGA algorithm uses a genetic algorithm whose genes of each single chromosome correspond with a slot from a specified resource. Due to the Grids environment which is a dynamic one, the MOGA algorithms decision-making takes place based on the static information of the slots. That is why, it cannot effectively optimize the performance. In addition, the user has no explicit control on allocating the resources to the tasks, thus this model cannot optimize the performance either. As a result, the allocation-cost and makespan of the proposed algorithm is less than both above-mentioned algorithms.

Fig. 3 reveals a comparison of runtimes of the three algorithms: the MinFCMT, the MOGA and the BE. The MinFCMT algorithm runtime shows an outstanding decrease, approximately three orders of magnitude relative to both the MOGA and the BE algorithms runtimes. The very low runtime of the MinFCMT algorithm is explained by the fact that the MinFCMT selects the best resource just once for each task, so its runtime will be very low. Whereas the MOGA algorithm uses genetic algorithm, that is, it needs to repetitively test a handful of the solutions in order

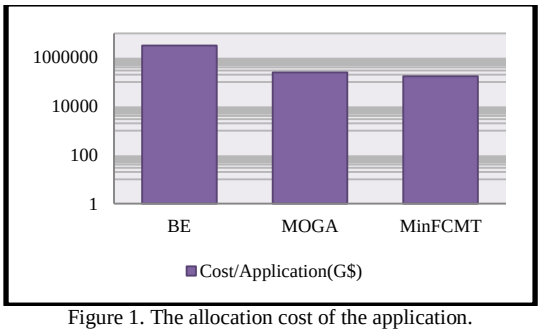


Figure 1. The allocation cost of the application.

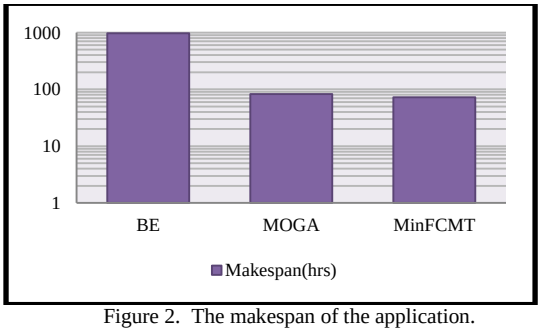


Figure 2. The makespan of the application.

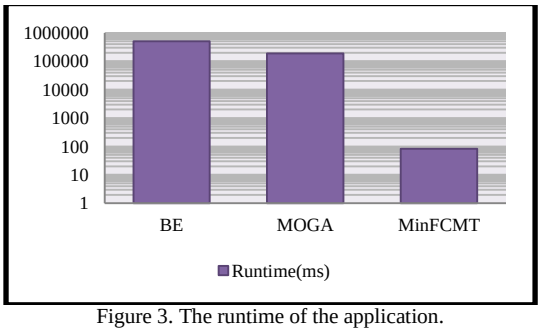


Figure 3. The runtime of the application.

to make decisions at each stage for planning the entire tasks. Therefore, this approach requires a high time-complexity to obtain the best solution. Compared to both the MinFCMT and the MOGA algorithms, the best-effort service has the highest runtime, because this approach does not supply the resources in advance. As a result, the proposed algorithm reveals a better performance compared to both algorithms.

5 Related works

As shared multiprocessing systems advance, the issue of the application scheduling has been the main concern. There is a comprehensive introduction on the job scheduling strategies [16, 17]. Moreover, in [18], the computational models are surveyed for Grid scheduling problems.

In the queue-based systems, the delivered quality of service to the users is the best effort QoS. However, the alternative approach is one of the planning-based systems [19]. In these systems, according to agreements the start time of the task can be established in advance instead of the task waits in queue in order to get access to the resource [20]. The above-mentioned agreements are based on an abstract description, so-called "slot".

In [21], a resource model is adopted similar to the proposed model. This model differs from the proposed model so that in this model, the resource provisioning takes place just for one task on a single resource, whereas in the proposed model, the resources are provisioned for the entire application tasks.

In [1], a heuristic algorithm is presented for scheduling many parallel applications on the Utility Grids that optimizes the cost-to-time trade-off. This approach is close to the studies conducted for this paper and its main difference from that of the proposed approach lies scheduling the parallel applications, whereas the approach adopted by present paper is based on scheduling the workflow application.

In [8], the Cloud computing for scheduling the scientific workflow is shown that uses the cost performance trade-off of the resource provisioning plans. This approach differs from the proposed approach, in that, it examines cost performance trade-offs disregarding the minimization of the multi-objective allocation-cost and makespan.

In [22], similar to the current paper, a genetic algorithm is proposed to find an optimized mapping of the tasks to the resources which minimizes both financial cost and makespan. This approach is developed in [14, 15], that is a multi-objective genetic algorithm. In [14, 15], the entire resources possess identical CPU ratings and cost processing whereas in the proposed model, all resources are constituted of the heterogeneous clusters with different processing cost and CPU ratings, that is the main difference between the proposed model and those models. Hence, removing this resource homogeneity complicates the identification of an appropriate resource selection.

Since, in [14, 15], the algorithms are genetic-based ones, the runtime takes a longer time. In case, the slots' characteristics undergo a change during scheduling, the slots' characteristics are to be updated and a rescheduled resulting in a far longer runtime. Hence, these approaches do not serve the purpose in the dynamic environments such as the Grids. In contrast, in the proposed approach, the WSC model is presented for the workflow scheduling problem which is highly adaptive to the dynamic environments. Also, this model is solved by the MinFCMT heuristic algorithm in an effective runtime.

6 Conclusion and future work

The present paper deals with designing, implementation and evaluation of the MinFCMT heuristic algorithm in order to schedule the applications. The paper seeks to optimize the cost-makespan based on the proposed WSC model. To develop a real distributed environment, the resources workload is simulated based on the Lublin model. Due to many experiments conducted on the generated syntactic workflow, it was shown that the MinFCMT heuristic algorithm is far more effective than the existing algorithms. The results indicate that the runtime of the MinFCMT algorithm decreases more than three orders of magnitude compared to the existing algorithms.

In future, we intend to develop an architecture by which the expected application performance can be determined with certainty and enhance our proposed algorithm.

7 References

[1] S. K. Garg, R. Buyya, and H. J. Siegel, "Time and cost trade-off management for scheduling parallel applications on Utility Grids," *Future Generation Computer Systems*, vol. 26, pp. 1344-1355, 2010.

[2] D. S. Katz, J. C. Jacob, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M. H. Su, and T. A. Prince, "A comparison of two methods for building astronomical image mosaics on a grid," in *proceedings of the 34th International Conference on Parallel Processing Workshops (ICPP 2005 Workshops)*, Oslo, Norway, 2005.

[3] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda, "GriPhyN and LIGO, building a virtual data grid for gravitational wave scientists," in *11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, Edinburgh, Scotland, UK, 2002.

[4] E. Deelman, S. Callaghan, E. Field, H. Francoeur, R. Graves, N. Gupta, V. Gupta, T. H. Jordan, C. Kesselman, and P. Maechling, "Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example," in *Proceedings of the Second IEEE international Conference on E-Science and Grid Computing* Amsterdam, Netherlands, 2006.

[5] R. Henderson, "Job scheduling under the portable batch system," in *Job Scheduling Strategies for Parallel Processing: IPPS '95 Workshop*, Santa Barbara, CA, USA, 1995, pp. 279-294.

[6] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a load sharing facility for large, heterogeneous distributed computer systems," *Software: Practice and Experience*, vol. 23, pp. 1305-1336, 1993.

[7] J. D. Ullman, "NP-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, pp. 384-393, 1975.

[8] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: the montage example," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, NJ, USA, 2008, pp. 1-12.

[9] J. Yu, R. Buyya, and C. K. Tham, "Cost-based scheduling of scientific workflow application on utility grids," in *First International Conference on e-Science and Grid Technologies (e-Science'05)*, Melbourne, Australia, 2005, pp. 140-147.

[10] R. Buyya and M. Murshed, "Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing," *Concurrency and Computation: Practice and Experience*, vol. 14, pp. 1175-1220, 2002.

[11] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger, "Data management in an international data grid project," in *Grid Computing - GRID 2000: First IEEE/ACM International Workshop*, Bangalore, India, 2000, pp. 333-361.

[12] U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: modeling the characteristics of rigid jobs," *Journal of Parallel and Distributed Computing*, vol. 63, pp. 1105-1122, 2003.

[13] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260-274, 2002.

[14] G. Singh, C. Kesselman, and E. Deelman, "An end-to-end framework for provisioning-based resource and application management," *Systems Journal, IEEE*, vol. 3, pp. 25-48, 2009.

[15] G. Singh, C. Kesselman, and E. Deelman, "A provisioning model and its comparison with best-effort for performance-cost optimization in grids," in *Proceedings of the 16th international symposium on High performance distributed computing*, Monterey, CA, USA, 2007, pp. 117-126.

[16] D. Feitelson and L. Rudolph, "Parallel job scheduling: Issues and approaches," in *1st Workshop on Job Scheduling Strategies for Parallel Processing*, Santa Barbara, CA, 1995, pp. 1-18.

[17] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *3rd Workshop on Job Scheduling Strategies for Parallel Processing*, Geneva, Switzerland, 1997, pp. 1-34.

[18] F. Xhafa and A. Abraham, "Computational models and heuristic methods for Grid scheduling problems," *Future Generation Computer Systems*, vol. 26, pp. 608-621, 2010.

[19] M. Hovestadt, O. Kao, A. Keller, and A. Streit, "Scheduling in HPC resource management systems: Queuing vs. planning," in *9th Workshop on Job Scheduling Strategies for Parallel Processing*, Seattle, WA, 2003, pp. 1-20.

[20] K. Czajkowski, I. Foster, and C. Kesselman, "Agreement-based resource management," *Proceedings of the IEEE*, vol. 93, pp. 631-643, 2005.

[21] T. Rblitz, F. Schintke, and J. Wendler, "Elastic Grid reservations with user-defined optimization policies," in *Proceedings of the Workshop on Adaptive Grid Middleware*, Antibes Juan-les-Pins, France, 2004.

[22] G. Singh, C. Kesselman, and E. Deelman, "Application-level resource provisioning on the grid," in *E-SCIENCE '06 Proceedings of the Second IEEE International Conference on e-Science and Grid Computing* Amsterdam, The Netherlands, 2006, pp. 83-83.

A Generic resources allocation approach for better Cloud Computing IaaS Services

EssamAlgizawy
Computer Engineering Department
Benha University
Shoubra-Cairo, Egypt
Essam@feng.bu.edu.eg

AlaaEldeenSayed Ahmed
Computer Engineering Department
Benha University
Shoubra-Cairo, Egypt
alaaeldeen.sayed@feng.bu.edu.eg

Abdulwahab K. Alsammak
Computer Engineering Department
Benha University
Shoubra-Cairo, Egypt
sammaka@gmail.com

Abstract *-The distributed Computation world is becoming complex and very large; Cloud Computing has emerged as a popular computing model to support large data processing using commodity hardware. In this context, the number of physical units (dedicated servers) may suffer shortage problem at peak time due to the increasing needs of using resources. In this paper we introduce a solution for resolving this problem by adopting the use of idle general purpose machines that exist within homes, universities or enterprises to be part of the cloud computing environment. The proposed solution is implemented and tested through the provision of the modified IaaS by using CPU cycles denoted by public and OpenStack Cloud Computing platform. On experimenting the proposed model ,the performance of the cloud has improve significantly ,which confirm that this mixing of generic and dedicated resources allocation leads to the provision of inexpensive resources with the guarantees of the quality of services.*

Keywords:Cloud Computing, CPU Cycle-Harvesting, Cloud Computing Infrastructure, Architecture

I. INTRODUCTION

The rapid growth in the distributed computing environment technologies has led away from the in-house Public Resources Computing to Grid computing and now to Cloud Computing paradigm. In this paradigm, Cloud resources are delivered in a flexible and service-oriented manner to consumers on an on-demand pay-as-you-go manner. This differs from common practice in IT whereby resources are procured and used in a more coarse-grained and less flexible manner. Cloud Computing is not a completely new concept for development and operations of the web application [3,12] .In the cloud we have to address a set of Cloud Computing fundamentals like virtualization management, scalability, interoperability, fail over mechanism and elasticity which represent one of the major attractions of the cloud. In Cloud Computing there are different delivery or deployment models such as Public, private, and Hybrid Clouds [13]. These models focus on the cloud architecture, and services as well. In Private Cloud, Data and processes are managed within organization boundaries without any restrictions on the network bandwidth, security exposures, or legal requirement for using service like these on public services. In Public Cloud, it describes the cloud computing in the traditional mainstream sense, whereby resources are dynamically provisioned on a fine grained, self-serviced basis over web services from an off-site third parity provider. Finally, in Hybrid Cloud, the environment is consists of two or private and public cloud computing providers.

Although private Cloud Computing is more secure than Public Cloud Computing, its elasticity is limited depending on the maximum hardware capacity and at the peak time it is bound to fall short of demand which is known as Cloud burst. As the needs for computational and storage resources increase the technology needs to develop to fulfil this increasing needs of using resources. We propose a new architecture to ad-hoc additional computational resources with reliable computing power using CPU cycles denoted by public (Organization PC.s) through the use of OpenStack Cloud computing software platform for on-premises Infrastructure as a Service (IaaS).

The availability of volunteer desktop machines as a part of the cloud leads to various benefits to individual organizations. It could reduce the numbers of dedicated machines needed to be purchased. Also it reduces the need for specialized infrastructure for resilience, such as redundant power and cooling systems, battery backup,… etc., which represents 25% of datacentre costs [8].Rather than ensuring resilience of a small number of physical buildings, the grain of resilience could be expanded by using more widely distributed machines and tolerating individual building failures [7]. As well as it reduces overall power consumption through the reduction in the total number of machines since the energy cost of manufacture for a computer has been estimated as four times that used during its lifetime [9].

The paper is organized as follows. Section 2 presents the related works on Cloud Computing IaaS implementations, Desktop Grid and public resources computing. The proposed Architecture is described in section 3. The architecture implementation and results is described in section 4. Finally, section 5 presents the conclusion.

II. RELATED WORK

The work presented in [14], explains an opportunistic cloud computing IaaS model implementation.It provides service at lower cost than dedicated cloud infrastructures based on basic computing resources (processing, storage and networking) through the opportunistic use of idle computing resources available in a university campus. UnaCloud uses a commodity and non-dedicated underlying infrastructure to implement the opportunistic design concepts which broadly has effect on; 1) the cloud services availability as the computing resources is dependent on the behavioural pattern of their currently owners, as the availability of the service represent an effective term to ensure any type of QoS or SLA. 2) Service scalability will always depend on the available resources on the opportunistic idle desktop machine which will limit the service scalability. 3) The architecture doesn't support integration with any of the currently used Cloud Computing platform APIs which limitsits use at peak times or at cloud burst.
Private cloud would provide flexible service availability and scalability while addition of hybrid clouds will providenot only elasticity but also would provideself-managing in terms of resilience, performance and balancing potentially.

In [25],an architecture is proposed to integrate grid and private cloud such that grid resource requirements are fulfilled by fetching resources from cloud when needed and vice versa with additional storage clusters to enhance the performance as Grid Computing resources are not utilized optimally most of the times.

In [7], an outline of the major implementation challenges has been introduced on how underused computing resources within an enterprise may be harnessed to improve utilization and create an elastic computing infrastructure, In contrast to the datacentre cloud model. This model is analogous to volunteer computing as exemplified by Condor [18] and BOINC [16, 17], although it poses considerable additional implementation challenges.

Volunteer computing (VC), Public Resources Computing (PRC) sometimes described as a paradigm of a large number of computers, on which individual users' machines are used to perform computationally tasks. SETI@home, Climateprediction.net, and LHC@home are the most popular projects based on the BOINC (Berkeley Open Infrastructure for Network Computing) framework [16, 17] or based on Condor [18]. Volunteer computing is being used in high-energy physics, molecular biology, medicine, astrophysics, climate study, and other areas.

In the context of IaaS models, OpenStack Cloud Computing platform [19, 20] is a collection of open source software projects that service providers can use to run their cloud compute and storage infrastructure. It is backed by Rackspace, NASA, Dell, Citrix, Cisco, and Canonical and over a hundred other organizations but it uses only dedicated servers to setup such Cloud Computing platform like that platform which created using Eucalyptus [21]

Our proposed approach focuses on extending the available Cloud Computing platform infrastructure which ensures service availability with the opportunistic use of idle resources which will provide services elasticity and service scalability through the using of both of the dedicated and non-dedicated infrastructures. It uses the same idea behind the building methodology of OpenStack Cloud Computing platform extending the infrastructure used by harvesting the idle computing resources donated by the public desktop machines.

III. PROPOSED ARCHITECTURE

In this approach we use the same idea behind the methodology of building the OpenStack Cloud Computing platform but with extending the used infrastructure by harvesting the idle computing resources donated by the public.
The architecture has three major components; Cloud Computing platform, Harvesting Middleware, and Resources allocation desktop agent.

1) ***Cloud computing platform:***The Cloud Computing platform consists of Cloud Controller which receives submitted user requests and passes them to the underlying components. It represents the interaction between computing nodes, block storage volumes, the networking controllers (software which controls network infrastructure), and API endpoints.

This interaction is represented in terms of managing and scheduling resources between different mentioned components. Since computing nodes have different operating systems running, there is virtualization machine manager called Hypervisor that mediates between the guest operating systems (Virtual machine or the Cloud Service) and the native hardware. The hypervisor is responsible to bring up virtualization by handling a set of certain protected instructions.

In our implementation we use Kernel Based Virtual Machine (KVM). KVM is a full virtualization solution that is unique in that it turns a Linux kernel into a hypervisor using a kernel module. This module allows other guest operating systems to then run in user-space of the host Linux kernel.

Each time a request for a new cloud computing services is submitted, the cloud controllers checks-up the available resources such as cores, memory, and storage to deploy the virtual machine. If there is a provisioning for deploying such virtual machine,the cloud controller will process the request by deploying the requested service. But if the virtual machine

can't be deployed on the current cloud infrastructure due to lack of resources, the cloud controller will ask the harvesting middleware local resources manager to free up resources. This is done by moving one of the currently running cloud services to the most suitable registered desktop machine.

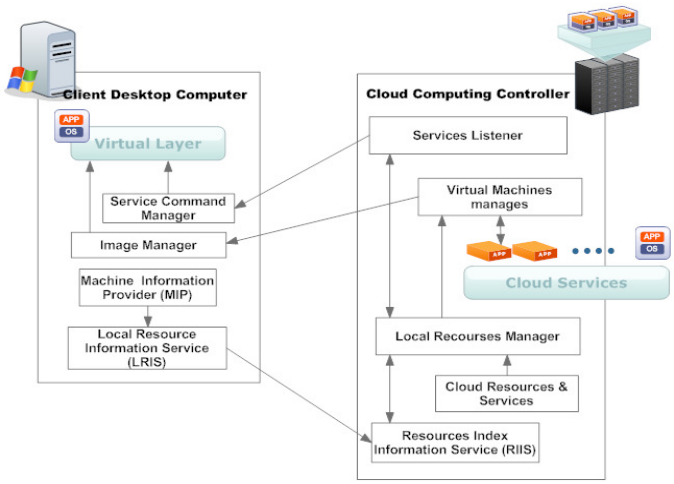


Figure 1 Harvesting Middleware Architecture

2) ***Harvesting Middleware:***The harvesting middleware consists of five daemons. Each one of these daemons plays a vital role in running any of the desktop machines included within the cloud computing. The five daemons are Resources index information service (RIIS), Cloud Resources and Service, Local Resources manager (LRM), Virtual machine manager (VMM), and Services Listener (SL) figure 1.

   a. Resources index information service (RIIS): RIIS represents a lightweight directory of the attached desktop machines, containing the basic information about local desktop resources and its aliveness status. This is done through a periodically heart beat message received from each registered desktop machine. RIIS collects information from several LRIS to allow searching the information to find the most suitable resources. The Desktop machine local resources information offered by the RIIS includes cores, memory, load status, desk storage and cloud service (Virtual machine) running on it if any.

   b. Cloud Resources and Service (CRS): it contains a complete list of Cloud platform resources e.g. computing node, network nodes, and storage. For each computing node there is a list of different running services associated with service information. This list may include, instance name, memory, virtualization type, kernel, desk source, interface type, service IP, DHCP server and all other information contained with a libvirt XML generated files.

   c. Local Resources manager (LRM): LRM is responsible for managing the execution of any cloud computing services on a desktop machine. It keeps track of all cloud computing services running on desktop machines, It also periodically query the status desktop running tasks from RIIS. If the Cloud Controller request the LRM to free up certain cloud resources, the LRM select the most suitable desktop machine from information provided by RIIS and CRS. Finally, it takes a snapshot of the running virtual machine to be deployed on the selected desktop machine.

   d. Virtual machine manager (VMM): VMM works in conjunction with the LRM to manage virtual machines in both the cloud environment and the desktop machines. It is responsible for creating image snapshots, suspending and starting running cloud services. Also, it can issue a deployment commends for starting , stopping , and monitoring cloud computing virtual machines on desktop PCs.

   e. Services Listener (SL): All users requests (Start, Terminate, Snapshot …etc.) are submitted to the cloud controller with the associated service identification. SL listen to all requests passed to the cloud controller and redirect basic requests to LRM registered cloud service running on desktop machines instead of passing it to either the compute nodes running these services or these that have a suspended image to guarantee service availability in case that desktop service failure.

3) ***Resources allocation desktop agent (RAD):***RAD consists of a simple four module, if we start from the bottom both of the Machine Information Providers (MIP) and Local Resources Information Service (LRIS) represent the resources discovery system which is responsible for publishing, and queuing the state of resources and their configurations. Up on installing the RAD agent on any desktop machine the MIP start by capturing information about the local resources which in turn passed to LRIS which represents the interface for RIIS on each desktop machine. Image Manager (IM) is responsible for issuing a simple virtualization commands for starting new virtual machine, terminating or monitoring. Also, it manages desktop virtual machine networking. The last module is the service command manager. It is used to execute basic requests redirect form SL Daemon on the running desktop cloud services.

IV. ARCHITECTURE IMPLEMENTATION AND RESULTS

In our implementation, the Cloud computing environment is established with the OpenStack [19,20] as a Cloud Computing middleware, KVM virtualization [6] as a hypervisor to

provide scalability and user isolation on OpenStack Cloud, and Libvirt [23] for hypervisor communication with cloud middleware.

The desktop Cloud services deployment includes the provisioning of the necessary data for its remote access using:

1) standard SSH [24] mechanisms
2) QEMUas a desktop virtualization tool [26]
3) Libvirt for virtualization interfaces.

Through the implementation we measure the values of the following parameters:

1) CPU utilization %: which depends on the software programs runningon this virtual machine, such as MySQL server or Apache server,… etc.
2) System Operation %: refers to the kernel's internal workings
3) IO requests wait%: is time spent waiting for I/O, such as a disk read or write as an important measurement parameter for the data intensive applications.
4) Memory used %: refers to the percentage of RAM used in performing tasks.

To test the previous parameters get results, we apply the following scenario:

1) Create Cloud computing services for monitoring various ways that CPUs spent on system performing different operations.
2) Monitor the system after the creation of cloud services without performing any user tasks whether it is by administration or by cloud client.
3) Load cloud services using a stress workload generator with half load and monitor system components performance.
4) Monitor cloud on peak load (peak usage time).
5) Suspend services on the cloud working node and monitor server performance.
6) Monitor number non-dedicated machined running windows 7 and display the average performance of all machines.
7) Move cloud service to desktop machine and record the machine performance.

By applying the previous scenario we recorded the following effects:

1) Effect on resources usages when creating N IaaS Cloud computing services.

From the client point of view, providing IaaS (Infrastructure as a service) is no more than creating a set of virtual machine by selecting the appropriate OS system required for the cloud client from those offered by the cloud provider. Upon creating service the client will have a virtual machine with a specific hardware specs running the chosen OS like any physical machines. IaaS Services/virtual machine creation consume almost all resources of the computing node host. The created virtual machine, effects on the performance of other VMs/services running on this host. Creating concurrent N service put the computing node under great stress as shown in fig 2. The Great stress effect is shown especially for the CPU and Memory utilization curves.

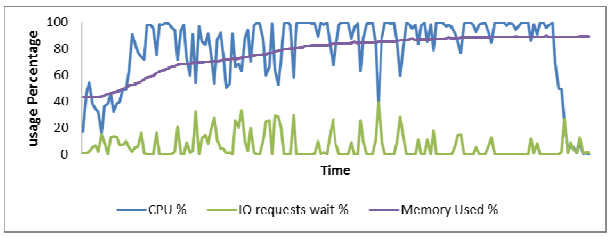


Figure 2 N Services (VM) Creation

2) Effect on resources usage rate after the time of creating N IaaS services and without loading any cloud services.

By monitoring different system parameters after IaaS services creation is done. We can see that about 90%of the computing node memory still used or reserved even through these services isn't using these resources in any useful work. The reason behind is that each virtual machine lock its own memory, which make it so difficult (consume longer time) to create another service on the same machine.

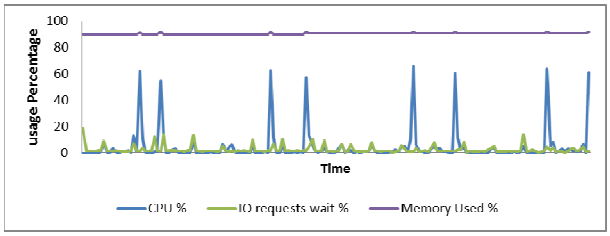


Figure 3 Cloud working node load after service creation without loading a cloud service

3) Effect on the resources percentage rate when half load running services occupied the system.

Upon putting the services under half load stress, , we can see that memory reach its maximum values and the CPU reach 90% at some points. System is fully utilized however the full load services are not reached yet.

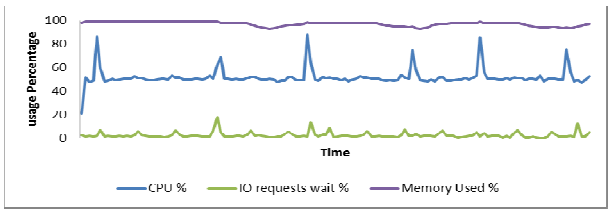


Figure 4Effect of Loading (N/2) of the running cloud services

4) Effect on the resources percentage rate when full load running services occupied the system.

Now, on full loading the cloud services, the computing node memory and CPU reach it maximum values which cause the system to reset a lot of tasks as we can see on fig 5. Not only but also the IO requests will wait for a significant amount of time. All these cause bad effect of the system throughput for the requested services.

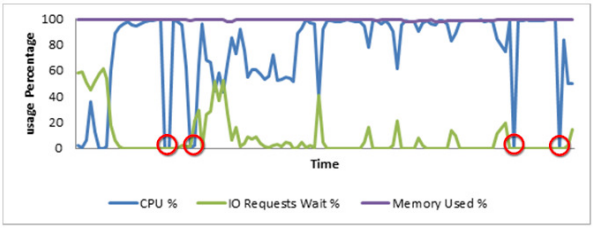


Figure 5 Effect of Loading N Cloud Service

5) Effect on computing node performance after suspending and moving service to a public desktop machine.

Now we apply our proposed solution by moving Cloud services (or part of the cloud load) to any available desktop machines (non-dedicated windows machines) with the same load. This movement is expected to increase the computing node performance due to decreasing its load and helping it to act well for the remaining part of the cloud services.

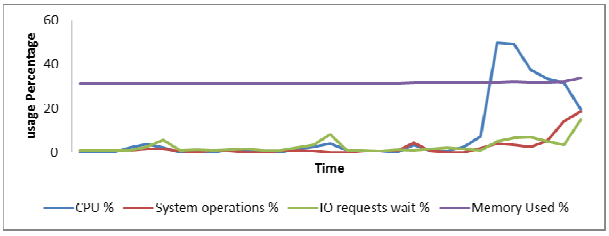


Figure 6 Cloud Computing node performances after suspending and moving service to desktop machine

6) Effect on desktop machine after completing the services movement process.

In normal cases we consume small part of our desktop machine computing power capabilities, as we can see on the following fig 7. Loading all services will have normal effect on desktop machine performance. As show in figure, over than 55% of machine memory is free to be used. Also, 80 % of CPU utilization is free to be used by desktop normal user. This indicates the good performance that we get when harvesting any available non-dedicated machines and use for freeing up the dedicated cloud computing nodes.

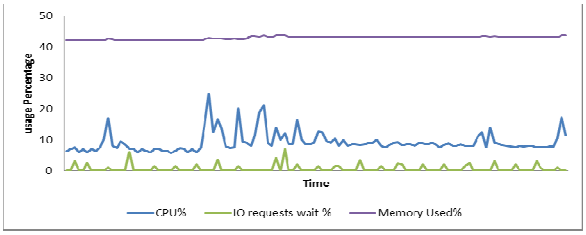


Figure 7 Regular usage desktop machine performances

7) Effect on performance when desktop and cloud computing users occupy the desktop machine,

The following two figures show the performance effect due to two cases. The first, when running IaaS service on the desktop (windows) machine while not running any other tasks need by the cloud computing our the desktop machine users. It still show the good performances showed in both memory and processor underutilization level. Memory is about 55% underutilized.

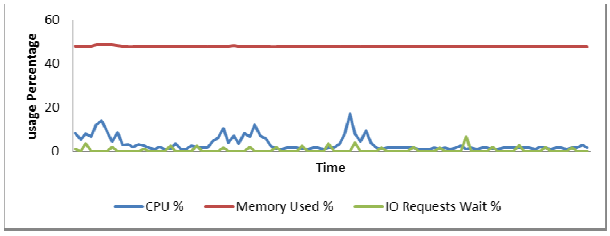


Figure 8 Run Cloud Service on windows machine (Both of the Client and service not loaded)

In the other hand, when user load services (whether it's by cloud user or desktop machine owner), the system is still working in the safe mode range. This mode will allow both users perform their tasks without affecting each other.

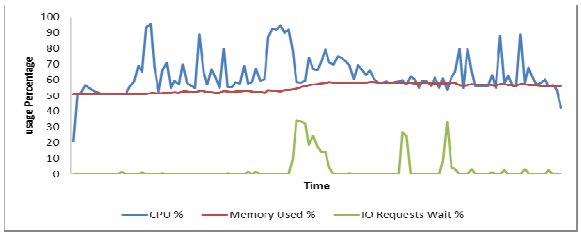


Figure 9 Run Cloud service on windows machine (Both of the Client and service loaded)

V. CONCLUSIONS

In this paper we presented a new approach which focus on extending the available Cloud Computing platform infrastructure to ensure service availability with the opportunistic use of idle resources which will provide services elasticity and service scalability through the using of both of the dedicated and non-dedicated infrastructure. Based on our implementation results, the action of moving cloud computing services to generic resources such as public desktop machines

easily available in homes, universities or enterprises will not affect the provided cloud services. On contrary it will help in improving the services and doesn't affect the public desktop machines performance at the same time.

VI. REFERENCES

[1] Ransome, John W. Rittinghouse& James F. *Cloud Computing: Implementation, Management, and Security*. s.l. : CRC Press / Taylor & Francis Group, 2010.

[2] Escalante, B. Furht& Armando, *Handbook of Cloud Computing*. s.l. : Springer Science and Business Media, 2010.

[3] Harris, torry. *Cloud Computing – An Overview*. s.l. :Torry Harris Business Solutions, Jan 2010.

[4] Foster, I., et al., et al. *Cloud Computing and Grid Computing 360-Degree Compared*. s.l.: Department of Computer Science, University of Chicago, Chicago, IL, USA, Jan 2009.

[5] OpenStack, Operating OpenStack Documentation. http://docs.stackops.org. [Online] 2012

[6] IBM. Virtual Linux. http://www.ibm.com. [Online] Dec 2006 .

[7] Dearle, A. Macdonald,A. Fernandes. *An Approach to Ad hoc Cloud Computing*. Graham Kirby, s.l.: University of St Andrews, 2010.

[8] Greenberg, A., Hamilton, J., Maltz, D. A. and Patel, P. *The Cost of a Cloud: Research Problems in Data Center Networks*.s.l. :ACM SIGCOMM Computer Communication Review, 2009. 10.1145/1496091.1496103.

[9] E.Williams. *Energy Intensity of Computer Manufacturing: Hybrid Assessment Combining Process and Economic Input-Output Methods*.s.l.: Environmental Science & Technology, 2004. 6166-6174.

[10] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, M. Tsugawa. Chicago,IL. *Science Clouds: Early Experiences in Cloud Computing for Scientific Applications*.: Cloud Computing and Its Applications, 2008.

[11] B. Sotomayor, K. Keahey,I. Foster. *Combining Batch Execution and Leasing Using Virtual Machines.* New York, NY, USA : ACM HPDC '08 Proceedings of the 17th international symposium on High performance distributed computing, 2008. 978-1-59593-997-5.

[12] Peng, Junjie, et al., et al. Shanghai .*Comparison of Several Cloud Computing Platforms.*: IEEE, Second International Symposium on Information Science and Engineering (ISISE), 2009. 978-1-4244-6325-1 .

[13] Rimal, B.P., Choi, Eunmi and Lumb, I. Seoul, South Korea. *A Taxonomy and Survey of Cloud Computing Systems.*: IEEE, INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference, Aug. 2009. 978-1-4244-5209-5 .

[14] E. Rosales, H. Castro, M. Villamizar. *UnaCloud: Opportunistic Cloud Computing Infrastructure as a Service.* Bogotá D.C., Colombia : IARIA, 2011. 978-1-61208-153-3.

[15] Amazon EC2. Amazon Elastic Compute Cloud (EC2) API Documentation . http://aws.amazon.com/. [Online] Amazon, 2012.

[16] California,University of. BOINC. http://boinc.berkeley.edu/. [Online] 2012.

[17] David P. Anderson, Eric Korpela , R. Walton. *High-Performance Task Distribution for Volunteer Computing.* University of California, Berkeley : IEEE , E-SCIENCE '05 Proceedings of the First International Conference on e-Science and Grid Computing, 2005. 0-7695-2448-6.

[18] D. Thain, T. Tannenbaum , M. Livny. *Distributed Computing in Practice: The Condor Experience.* 2-4, s.l.: Concurrency and Computation: Practice & Experience, February 2005, Vol. 17.

[19] OPenStack. Documentations. http://docs.openstack.org/. [Online] 2012.

[20] OpenStack. Nova's documentation. http://nova.openstack.org/. [Online] 2012.

[21] Eucalyptus. Eucalyptus Cloud Computing Documentation. http://www.eucalyptus.com. [Online] 2011.

[22] RabbitMQ. Documentation. http://www.rabbitmq.com. [Online] 2011.

[23] IRC, OFTC. Documentation. http://libvirt.org/. [Online] 2011.

[24] The Apache Software Foundation. Documentation. http://mina.apache.org. [Online] 2011.

[25] K. Selvaraj, S. Mukherjee.*Integration of Grid and Cloud with Semantics based integrator.* Centre for Dev. of Adv. Comput. (CDAC), Chennai, India: Computational Intelligence, Communication Systems and Networks (CICSyN), 2011 Third International Conference on, August 2011 . 978-0-7695-4482-3.

[26] QEMU. Documentation. *http://wiki.qemu.org*. [Online] 2011.

CDAC Scientific Cloud: On Demand Provisioning of Resources for Scientific Applications

A. Payal Saluja, B. Prahlada Rao B.B, C. Ankit Mittal and D. Rameez Ahmad
SSDH, Centre for Development of Advanced Computing
C-DAC Knowledge Park, Bangalore, Karnataka, India

Abstract - Scientific applications have special requirements of availability of a massive computational power for performing large scale experiments and huge storage capacity to storage terabyte or petabyte range of outputs. Scientific Cloud provides scientists computational, storage and network resources with a inbuilt capability of utilizing the infrastructure. The scientific applications can be dynamically provisioned with the required cloud solutions that are tailored to the application needs. Centre for Development of Advanced Computing (CDAC) under Department of IT, is the pioneer in HPC in India with ~70TF compute power. The authors of this paper have discussed the need and benefits of scientific cloud. Authors have explained the model, architecture and components of CDAC scientific cloud. CDAC HPC resources can be provisioned on-demand to the scientific research community and released when they are not required. For Indian researchers and scientists, CDAC scientific cloud model will provide convenient access to reliable, high performance clusters and storage, without the need to purchase and maintain sophisticated hardware.

Keywords: HPC, HPC as a Service, Map Reduce, Cloud Vault

1 Introduction

High Performance Computing (HPC) allows scientists and engineers to solve complex science, engineering and business problems using applications that require high bandwidth, low latency networking, very high compute and storage capabilities. Scientists in the areas of high-energy physics [13], astronomy [14], climate modeling [15], chemo informatics [16] and other scientific fields, require massive computing power to run experiments and huge data centers to store data. Typically, scientists and engineers must wait in long queues to access shared clusters or acquire expensive hardware systems.

Cloud computing [17] is a model for on-demand access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, services, and software) that can be easily provisioned as and when needed. Cloud computing aggregates the resources to gain efficient resource utilization and allows scientists to scale up to solve larger science problems. It also enables the system software to be configured as needed for individual application requirements.For research groups, cloud computing will provide convenient access to reliable, high performance clusters and storage, without the need to purchase and maintain sophisticated hardware. It has been said by Pete Beckman, director of Argonne's Leadership Computing Facility that "Cloud computing has the potential to accelerate discoveries and enhance collaborations in everything from providing optimized computing environment for scientific applications to analyzing data from climate research, while conserving energy and lowering operational costs". However, there are various challenges of HPC on demand [29] like performance, power consumption and collaborative work environments. In this approach paper, we present the concept of scientific clouds, HPC as a service and its benefits to the scientific research community. The authors also propose a prototype for CDAC scientific cloud that will provide the following offerings

I. Infrastructure as a Service(IaaS)[18] by providing traditional MPI enabled HPC with parallel file system like GlusterFS[19] and by provisioning Hadoop[20] clusters with map reduce[21] with the support of Hadoop distributed file system(HDFS)[20].
II. Storage as a service (StaaS) [22] to provide petabytes of data storage to the scientific communities.

The rest of the sections of this paper are organized as follows: Section 2 describes the concept of HPC as a Service, the challenges of HPC on cloud and how cloud computing benefits the scientific community. Section 3 talks about the other scientific cloud projects and their objective and the relevant work. Section 4 details about the CDAC scientific cloud and its offerings. Section 5 details the proposed model and architecture for the CDAC scientific cloud. Section 6 talks about the applications that will be enabled on CDAC scientific cloud. Section 7 concludes and tells about the future plan of the work.

2 HPC as a Service on Cloud

Bringing HPC facilities to cloud will provision the scientists and researchers with a crucial set of resources and enable them to solve large-scale, data-intensive, advanced computation problems on research topics across the disciplinary spectrum. HPC as a service is an on-demand provisioning of high-performance, scalable HPC environment with high-density compute nodes and huge storage on high performance interconnects like Infiniband [4] and Myrinet [5]. HPC as a service is provisioned to meet the HPC application demands, whether one server (Virtual machine) or a large cluster (Virtual cluster). A Virtual cluster is a collection of Virtual Machines configured to interact with each other as a traditional Linux cluster. Scientific cloud or HPC as a Service enables greater systems flexibility and eliminates the need for dedicated hardware resources per applications and would help researchers cope with exploding volumes of data that need to be analyzed to yield meaningful results. It also simplifies usage models and enables dynamic allocation per given task.

[6] Described a demonstration of a low-order coupled atmosphere-ocean simulation running in parallel on an EC2 system. It highlights the significant way in which cloud computing could impact traditional HPC computing paradigms. The results show that the performance is below the level seen at dedicated, supercomputer centers, however, performance is comparable with low-cost cluster systems. Also it has been concluded that it is possible to envisage cloud systems more closely targeted to HPC applications, that feature a specialized interconnect such as Myrinet or Infiniband.

Scientific Cloud benefits to the Scientists & research Community:

- **Dynamic Provisioning of HPC Clusters**: Access to on-demand cloud resources enables automatic provisioning of additional resources from the HPC service to process peak application workloads, reducing the need to provision data center capacity according to peak demand. Hence, scientists will benefit from the ability to scale up and down the computing infrastructure according to the application requirements and the budget of users.
- **Virtual ownership of resources** : Virtual ownership of cloud resources will reduce uncertainty concerning access to those resources when you need to use them
- **Ease of deployment and access**: The use of virtual machine images offers the ability to package the exact OS, libraries, patches, and application codes together for deployment. Scientists can have easy access to large distributed infrastructures and completely customize their execution environment, thus providing the perfect setup for their experiments.
- **Reduction in overall Job execution time**: Jobs will be scheduled using intelligent data aware job scheduling algorithms.

Figure 1 depicts the layered architecture of scientific cloud. The lowest layer of the stack is the physical resources (compute, storage and network) that will be connected through a high speed link. The first software layer above the physical hardware is the host operating system. Since scientific cloud will be catering HPC applications, performance of such applications on such infrastructure will be of prime importance. Hence, Type 1 or bare-metal hypervisor should be preferred for virtualization that will run directly on the host's hardware to control the hardware and to manage the guest operating systems.

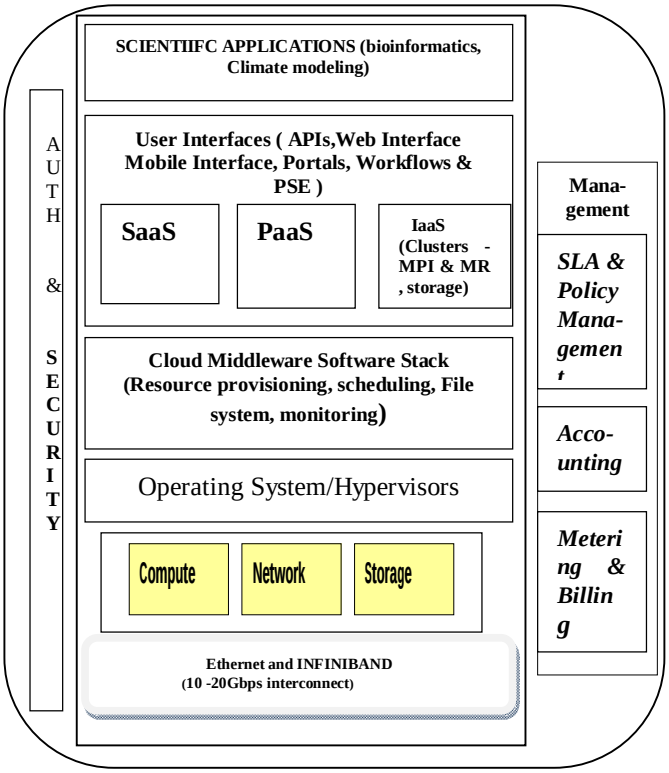


Figure 1 Scientific Cloud Architecture

A guest operating system will run on another level above the hypervisor. Hypervisor actually controls the host processor and resources, allocating what are needed to each operating system in turn and making sure that the guest operating systems (called virtual machines) cannot disrupt each other. The virtualized resources include the basic cloud computing services such as processing power, storage, and network. The Cloud middleware software stack is the key component that handles resource provisioning and scheduling, volume management, system monitoring for all the higher-level components and services.

Cloud management is a crucial component as it monitors and manages all the cloud resources at physical and virtual level. The various management components that will be part of scientific cloud are : Resource Inventory search, Hardware monitoring & Management , Storage maps and reports, Alerts & notifications with automated rectification, accounting and billing(to recover costs, capacity planning to ensure that consumer demands will be met) , policy management & SLA(Service level Agreements- management to ensure that the terms of service agreed to by the provider and consumer are adhered to, and reporting for administrators).

3 Science Cloud Projects

The following are some of the science cloud projects have been executed in the direction to achieve HPC as a Service:

3.1 Cumulus

Cumulus [2] is a project to build a Scientific Cloud for a Data Center. It is a storage cloud system that adapts existing storage implementations to provide efficient upload/download interfaces compatible with S3.It provides features such as quota support, fair sharing among clients, and an easy to- use, easy-to-install approach for maintenance. The most important feature of Cumulus is its well-articulated back-end extensibility module. It allows storage providers to configure Cumulus with existing systems such as GPFS [9], PVFS [10], and HDFS [11], in order to provide the desired reliability, availability or performance trade-offs. Cumulus is part of the open source Nimbus toolkit [12]. Cumulus is implemented in the python programming language as a REST service. The Cumulus API is a set of python objects that are responsible for handling specific user requests.

3.2 OpenCirrus

Open Cirrus [3] tested is a collection of federated datacenters for open-source systems and services research. It is designed to support research into the design, provisioning, and management of services at a global, multi-datacenter scale. It is designed to encourage research into all aspects of service and datacenter management.

3.3 GridGain

GridGain[30] is Java based open source middleware for real time big data processing and analytics that scales up from one server to thousands of machines. It enables the development of compute and data intensive High Performance Distributed Applications. Applications developed with Gridgain can scale up on any infrastructure - from a single Android device to a large cloud. Gridgain provides two major functionalities of Compute Grids and In-Memory Data Grids

3.4 StratusLab

Stratus Lab [31] is developing a complete, open-source cloud distribution that allows grid and non-grid resource centers to offer and to exploit an Infrastructure as a Service cloud. It basically enhances the grid infrastructure with virtualization and cloud technologies. It is particularly focused on enhancing distributed computing infrastructures such as the European Grid Infrastructure (EGI).

Each of the above mentioned projects focuses either on provisioning data centers on cloud or compute power on cloud. **Amazon Web Services** alone provisions the various services required for variety of HPC applications like Amazon Elastic Compute cloud EC2, Amazon Elastic Map Reduce (EMR), Amazon Simple Storage Service (S3) [32]. **CDAC Scientific cloud** is an effort to provide the services like compute and storage for the HPC community along with the software technologies like map reduce, MPI , mobile applications that will accelerate discoveries and enhance collaborations in science.

4 CDAC Scientific Cloud (CSC)

C-DAC [7] is the pioneer in HPC in India and its HPC facilities on cloud can be linked by a 1 Gbps National Knowledge Network (NKN) [8], developed by NIC. The bandwidth offered by NKN will facilitate rapid transfer of data between geographically dispersed clouds and enable scientists to use available computing resources regardless of location. In addition, CDAC Scientific cloud will provide data storage resources that will be used to address the challenge of analyzing the massive amounts of data being produced by scientific applications and instruments. Storage as a service is of particular importance to scientific research, where volumes of data produced by one community can reach the scale of terabytes per day .CDAC will make the Scientific cloud storage available to science communities by aggregating a set of storage servers .It will make use of advanced technologies to provide fast random access storage to support more data-intensive problems. The test bed will be a mix of virtual clusters and storage options, traditional HPC cluster, Hadoop cluster, distributed and global disk storage, archival storage. The system provides both a high-bandwidth, low-latency InfiniBand network as well as a commodity Gigabit Ethernet network. This configuration is different from a typical cloud infrastructure but is more suitable for the needs of scientific applications.

Using CDAC Scientific cloud instances, users can expedite their HPC workloads on elastic resources as needed .Users can choose from Cluster Compute or Cluster Hadoop instances within a full-bisection high bandwidth network for tightly-coupled and IO-intensive workloads or scale out across thousands of cores for throughput-oriented

applications. This will let scientists focus on running their applications and crunching or analyzing the data generated by applications without having to worry about time-consuming set-up, management or tuning of clusters or storage capacity upon which they sit. Users will be able to run HPC applications on these instances including molecular modeling, genome sequencing & analysis, and numerical modeling across many industries including Biopharma, Oil and Gas, Financial Services and Manufacturing. In addition, academic researchers will be able to perform research in physics, chemistry, biology, computer science, and materials science.

Following will be the supported features of CDAC High Performance Computing as a service (HPCaaS):

- ✓ **Dynamic Provisioning of clusters** :On demand Provisioning MPI and Map reduce clusters to support compute intensive and data intensive applications
- ✓ **On-demand dynamic provisioning of storage volumes**: Dynamic provisioning of clusters and storage will be handled by the Cloud Resource Broker (CRB) or cloud metaschedular.
- ✓ **Security** : Simple , Secure and quick access to HPC clusters
- ✓ **Provisioning of customized libraries, softwares** workflows ,etc on HPC clusters as per the applications requirement Users will be provided with an option of selecting the specific MPI versions or compiler versions to suffice the application requirements
- ✓ **Performance:** To reduce the hypervisor overhead type-1 kind of hypervisor will be used. The distributed locations will be connected with 1Gbps link and within the site nodes will be connected with infiniband interconnect to reduce the latencies.VM allocation to form a cluster will be done by the cloud scheduler based on nearness to storage nodes to minimize the data movement on cloud.

Following are the services that will be provisioned on the CDAC Scientific Cloud

4.1 Infrastructure as a Service (IaaS)

C-DAC has its HPC facilities at various CDAC locations like Bangalore, Pune, Chennai, and Hyderabad with approximately 70TF. Figure 2 depicts the prototype model for dynamic provisioning of the computational resources when requested by the user. Users will be able to access the CDAC scientific cloud services through cloud portal. First time users will have to register with their required details and also the details about the kind of applications they want to run on the cluster. Based on the type of the application mentioned by the user resources will be allocated by the cloud broker and the cluster instance will be created on the fly. Immediately user will be intimated online and through mail about their login credentials and the IP address for the ssh access to the compute cluster. The allocation of the cluster and its nodes (master & worker nodes) will depend upon the CPU, memory, IO requirements of the application. The applications that will need more of data processing and less of communications will provided with the best suited map reduce cluster. The applications that are more compute and memory intensive will be provisioned by the MPI enabled clusters with parallel IO facility.

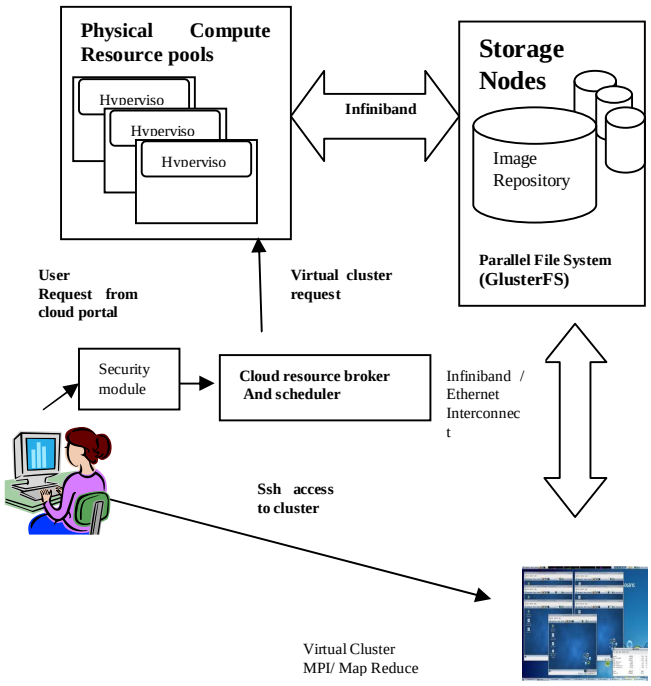


Figure 2 Infrastructure as a Service (IaaS)

4.2 Storage As a Service (StaaS)

A service of supplying data storage capacity over Internet is Storage as a Service. In context of scientific cloud, StaaS provisions petabytes of data storage to the scientific communities. CDAC's Cloud Vault based on OpenStack Swift Object Storage software will provide scientists and researchers partners with a convenient and affordable way to store, share, and archive data, including extremely large data sets. CDAC Cloud Vault is an object based storage system and multiple interface methods make the Cloud Vault easy to use for the average user. It also provides a flexible, configurable, and expandable solution to meet the needs of more demanding applications. In this, files (also known as objects) are written to multiple physical storage arrays simultaneously, ensuring at least two verified copies exist on different servers at all times. Figure 3 depicts the flow of the Storage as a service (StaaS).The user registers himself by

providing the required details and the required amount of storage. After the users request gets validated and approved, user is sent the access details of the storage through email. The various interfaces through which user can access Cloud Vault are as follows:

4.2.1 Web Interface

Web interface will allow access to the cloud vault files through browser. User will be able to list, create containers, Upload/Download files, and Delete files using this interface. There will not be any need of installation of any clients to access cloud files.

4.2.2 Desktop GUI Application

Cloud Vault files will be accessible using open source desktop application called cyberduck. It is an FTP-like stand alone GUI application for accessing files. It supports file/ directory listing, upload, download, synchronize, editing, etc. Cyberduck is a open source desktop application available for MAC and Windows system

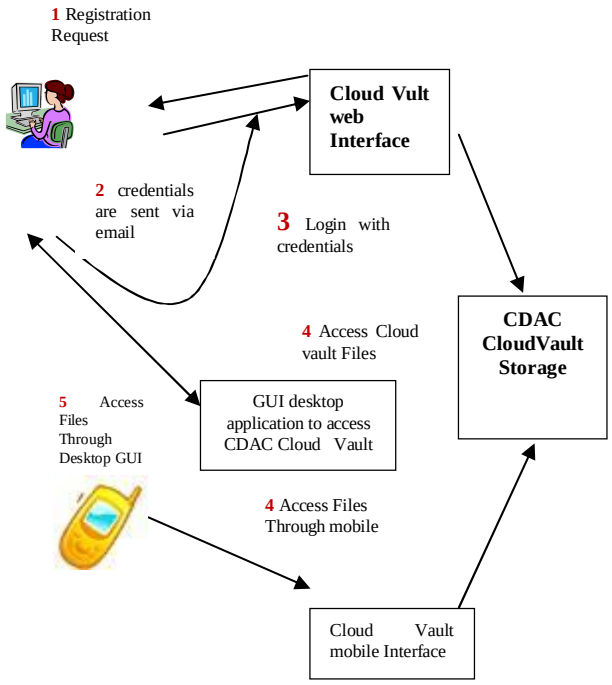


Figure 3 Storage as a Service (StaaS)

4.2.3 Command Line

Command line access will allow the access to cloud vault files with the UNIX shell. Client installation of the scripts needs to be done on the user machine or laptop.

4.2.4 Mobile Interface

Cloud Vault will also be accessed by mobiles using mobile application for the basic file operations like list, upload, download, and synchronize. There will also be a facility to auto synchronize users mobile with his cloud vault files so that he can keep his mobile backup on cloud vault.

4.2.5 APIs

Files of any size can be stored in the Cloud Vault, from small personal document collections to multi-terabyte backup sets routed directly to the cloud using Rack space or S3 API in applications.

5 CSC Architecture and Components

Figure 4 depicts the components of CDAC scientific cloud. The various components of CDAC scientific cloud are as follows:

5.1 Hypervisor

A hypervisor, also known as a virtual machine manager/monitor (VMM), is computer hardware platform virtualization software that allows several operating systems to share a single hardware host. The hypervisor controls the host processor and resources so that systems/virtual machines are unable to disrupt each other. As virtualization adds overheads to the cluster performance, we choose to use type-1 or bare-metal hypervisors for virtualization. Type-1 hypervisors run directly on the host's hardware to control the hardware and to manage guest operating systems. Some of the examples of type-1 hypervisors are Citrix XenServer [24], VMware ESX/ESXi [25], and Microsoft Hyper-V hypervisor. CDAC scientific cloud will be using Xen hypervisor for the same.

5.2 Cloud middleware

Cloud Middleware or Cloud OS: Cloud middleware is the software stack for provisioning the large networks of virtual machines on demand. It also handles scalability & reliability of the resources provided to the users. There are various open source & commercial cloud middleware available like Nimbus [12], Open Nebula [26], and vCentre [27], Eucalyptus [28].

5.3 Cloud resource broker

Cloud Resource Broker and Meta scheduler: Cloud resource broker is a common gateway to provision access to the HPC resources like compute clusters, storage on cloud .It is an intelligent scheduler that will provision the best pool of available resources to the users by using policy based decision. The various components that will build up a cloud resource broker are as follows:

5.3.1 Resource Discovery

Resource discovery of the available resources based on the kind of user application that can be Compute intensive or Data intensive or Memory Intensive

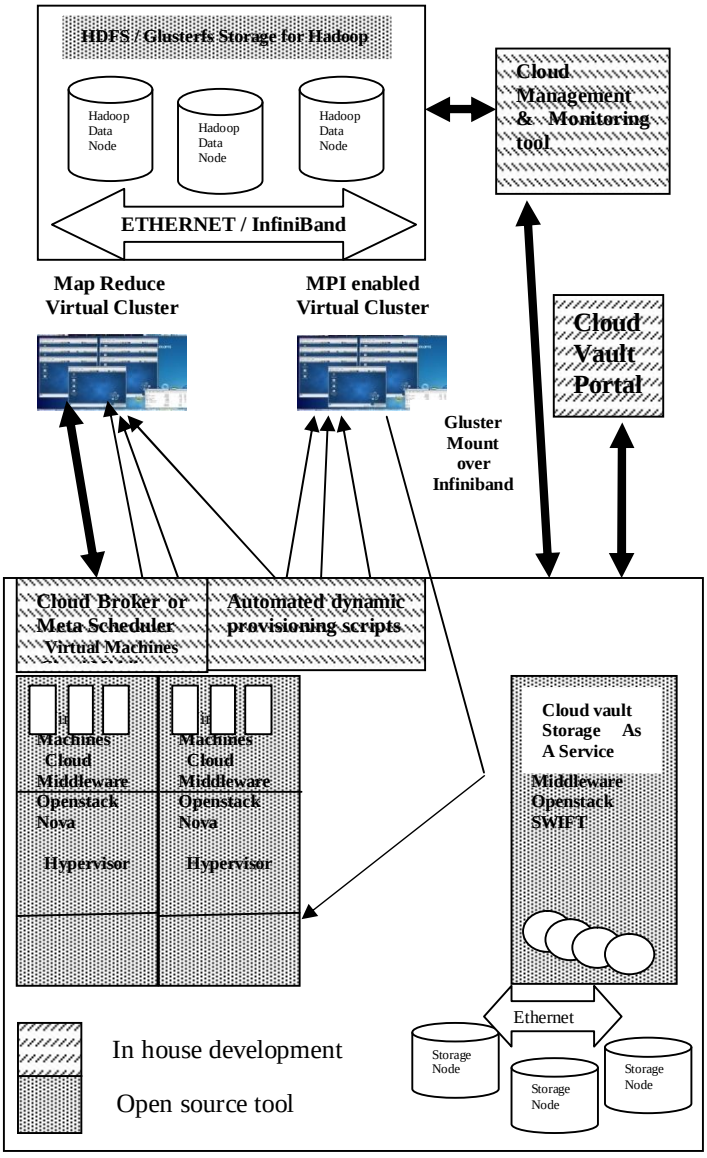


Figure 4 Components of CDAC Scientific Cloud (CSC)

5.3.2 Policy based resource selection

Resource selection and provisioning will be done considering the various aspects like Load balancing, resources utilization, power aware.

5.3.3 Data aware Job scheduling

Data aware scheduling enables computation to be done nearest to the location of the data .In this case, the cloud resource broker will talk to the cloud file system components to find out the nearest storage nodes where data resides

5.4 Cloud Management and Monitoring

Cloud Infrastructure monitoring & management tool is the control point for the virtual environment in cloud. This tool will provide a single point access for administrators to monitor & manage the resources of cloud. The following features that will be supported :

- Resource Inventory search: inventory including virtual machines, hosts, data stores, and networks at the administrators fingertips from anywhere
- Hardware monitoring & Management
- Storage maps and reports: Provides storage usage, connectivity and configuration. Customizable topology views give you visibility into storage infrastructure and assist in diagnosis and troubleshooting of storage issues.
- Alerts & notifications with automated rectification
- Utilisation, Performance & Energy Consumption Trends
- Accounting and billing (to recover costs, capacity planning to ensure that consumer demands will be met), Policy management & SLA.

5.5 Cloud Portal

5.5.1 Portal for IaaS Provisioning and Problem Solving Environments (PSE)

The scientific cloud portal will be the access point for the users for requesting & accessing the on demand HPC clusters. There will also be customized PSEs for bioinformatics & climate modeling domains that will provide the complete environment and workflow for the domain specific applications.

5.5.2 Portal for Storage as a Service

The portal for storage as a service will an access point for the cloud storage through which user can register himself and ask for the required amount of storage .Also user will be allowed to request for expanding the allocated storage on the fly

6 Target Applications on CDAC Scientific Cloud

On-demand cloud computing can add new dimension to HPC, in which virtualized resources can be sequestered, in a form customized to target a specific application requirement, at any point of time. [6] Described the feasibility of running Coupled Atmosphere-Ocean Climate Models on an EC2 computing cloud and found that the performance is below the level seen at dedicated clusters. However, cloud systems that feature a specialized

interconnect such as Myrinet or Infiniband and support MPI or Map reduce are more closely targeted to HPC applications.[23] states that Life Sciences are very good candidates for Map Reduce on cloud including sequence assembly and the use of BLAST and similar algorithms for sequence alignment. On the other hand partial differential equation solvers, particle dynamics and linear algebra require the full MPI model for high performance parallel implementation on cloud. The two application domains that have been identified as pilot applications for CDAC scientific cloud are Bioinformatics applications like Blast, Climate Modeling like Seasonal Forecast model (SFM). Seasonal Forecast Model (SFM) is an atmosphere general circulation model used for predicting the Indian summer monsoon rainfall in advance of a season. It involves the single operation on multiple data sets that makes it a suitable case for using map reduce in this particular application

7 Conclusions and Future Plans

Scientific applications require the availability of massive compute and storage resources. Cloud computing can be of great help in on demand provisioning of the HPC resources. The applications can scale up heavily using HPC as a service on cloud. However, the performance related challenges have to be addressed by fine tuning the cloud middleware stack and the software libraries. The proposed model of CDAC scientific cloud is an attempt to address the requirements and challenges of HPC as a service on cloud. Currently, the test bed setup for the same is in progress and in future we plan to develop the cloud system software components like Cloud Resource Broker and Meta scheduler, management and monitoring tools, portal & PSEs

8 References

[1] K. Keahey1, R. Figueiredo2, J. Fortes2, T. Freeman1, M. Tsugawa2, Science Clouds: Early Experiences in Cloud Computing for Scientific Applications, 1University of Chicago, 2University of Florida

[2] Cumulus: John Bresnahan, David LaBissoniere http://www.nimbusproject.org/files/bresnahan_sciencecloud2011.pdf

[3] Roy Campbell,5 Indranil Gupta,et. Al, Open CirrusTM, Cloud Computing Testbed: Federated Data Centers for Open Source Systems and Services Research,.

[4] http://en.wikipedia.org/wiki/InfiniBand

[5] http://en.wikipedia.org/wiki/Myrinet

[6] Constantinos Evangelinos and Chris N. Hill, Cloud Computing for parallel Scientific HPC Applications: Feasibility of running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2, CCA-08 in Chicago

[7] www.cdac.in

[8] www.nkn.in

[9] http://www.darwinproject.mit.edu/wiki/images/2/2e/Gpfs_overview.pdf

[10] Philip H. Carns, Walter B. Ligon III, Robert B. Ross Rajeev Thakur, PVFS: A Parallel File System for Linux Clusters, In Proc. of the Extreme Linux Track: 4th Annual Linux Showcase and Conference, October 2000

[11] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, http://moodle.openfmi.net/file.php/331/lectures/lecture_4/The_Hadoop_Distributed_File_System.pdf

[12] The Nimbus Toolkit: www.nimbusproject.org

[13] http://www.nersc.gov/assets/HPC-Requirements-for-Science/Spentz.pdf

[14] http://www.stfc.ac.uk/resources/pdf/ctreport.pdf

[15] http://www.mmm.ucar.edu/events/indo_us/PDFs/0630_SKDash_HPC-USA-final.pdf

[16] http://www.daylight.com/cheminformatics/casestudies/infinity.html

[17] http://www.gartner.com/it-glossary/cloud-computing/

[18] http://searchcloudcomputing.techtarget.com/definition/Infrastructure-as-a-Service-IaaS

[19] http://www.gluster.org/about/

[20] Hadoop, http://en.wikipedia.org/wiki/Apache_Hadoop

[21]]Map Reduce, http://hadoop.apache.org/common/docs/current/mapred_tutorial.html

[22] http://searchstorage.techtarget.com/definition/Storage-as-a-Service-SaaS

[23] http://grids.ucs.indiana.edu/ptliupages/publications/CloudsandMR.pdf

[24] Citrix Xenserver, http://www.citrix.com/English/ps2/products/product.asp?contentID=683148

[25] VMWare ESXi, http://www.vmware.com/files/pdf/VMware-ESX-and-VMware-ESXi-DS-EN.pdf

[26] OpenNebula:http://opennebula.org/

[27] vCentre, http://www.vmware.com/products/vcenter-server/overview.html

[28] Eucalyptus, http://www.eucalyptus.com/

[29] http://www.penguincomputing.com/files/whitepapers/PODWhitePaper.pdf

[30] http://www.gridgain.com/features/

[31] http://stratuslab.eu/doku.php/start

[32] http://aws.amazon.com/hpc-applications/

SESSION

SCHEDULING ALGORITHMS AND JOB SCHEDULING + LOAD-BALANCING + APPLICATIONS

Chair(s)

TBA

Exploiting Instruction Level Parallelism for REPLICA – A Configurable VLIW Architecture with Chained Functional Units

Martin Keßler[1], Erik Hansson[1], Daniel Åkesson[1], and Christoph Kessler[1]
[1]Dept. of Computer and Information Science, Linköping University, Sweden

Abstract— *In this paper we present a scheduling algorithm for VLIW architectures with* chained functional units. *We show how our algorithm can help speed up programs at the instruction level, for an architecture called REPLICA, a configurable emulated shared memory (CESM) architecture whose computation model is based on the PRAM model. Since our LLVM based compiler is parameterizable in the number of different functional units, read and write ports to register file etc. we can generate code for different REPLICA architectures that have different functional unit configurations. We show for a set of different configurations how our implementation can produce high quality code; and we argue that the high parametrization of the compiler makes it, together with the simulator, useful for hardware/software co-design.*

Keywords: Scheduling, VLIW, Compiler, LLVM, instruction-level parallelism

1. Introduction

The REPLICA architecture, which is currently under development, is a chip multiprocessor with configurable emulated shared memory (CESM) architecture [1]. Its computation model is based on the PRAM (Parallel Random Access Machine) model [2]. The PRAM model gives a simple deterministic synchronous and predictable model of programming, where parallelism is homogeneous and explicit. The REPLICA core architecture is a VLIW architecture that supports chained functional units so that the result of one VLIW sub-instruction can be used as input to another sub-instruction in the same (PRAM) execution step. The architecture has support for parallel multi-(prefix)operations on the hardware level [3]. It enables the user to access the same memory location from a multitude of parallel threads. There is no need for explicit locking as it would be in conventional parallel programming. By running a high number of threads in parallel, latency at memory accesses is effectively hidden by the architecture. Shared (physically distributed across on-chip memory modules) memory is in PRAM mode accessed in a UMA (Uniform Memory Access) fashion as if it was local memory.

REPLICA also has support for NUMA mode execution, for being able to run sequential and parallel legacy (NUMA) programs.

At the moment a high-level programming language for REPLICA is under development and should be transformed to the REPLICA baseline language using a source-to-source transformer which is currently under development using ANTLRv3 and written in Scala [1].

The baseline language is based on C with some built-in variables to support parallelism, at the moment these are for example `_thread_id`, `_number_of_threads` and `_private_space_start`.

Programs written in the baseline language can be compiled using Clang to LLVM IR and then compiled to REPLICA target code and tested and evaluated on the REPLICA simulator. The key feature of the compiler is the parametrization of the scheduling algorithm. In an earlier version of the compiler [4] there was only support for one basic architecture configuration.

The focus in this paper is to show how the compiler actually utilizes instruction level parallelism for different configurations of the REPLICA architecture in which we have different combinations of chained functional units.

As a proof of concept we have written, in the baseline language, some test programs such as thread parallel blur and threshold filter as well as sequential programs such as multiply & move, discrete wavelet and inverse discrete cosine transformation. We compiled them for different configurations using our parametrized back-end. The compiled programs are run on the simulator. They all show speed-ups from instruction level parallelism.

The rest of the paper is organized as follows. We introduce the REPLICA assembly programming model in section 2. We give an overview of how the dependency graphs of basic blocks are created in section 3, and in section 4 a method of reducing register use is shown. In section 5 we present our scheduling algorithm, and how the compiler is parametrized is shown in section 6. We compare our scheduling algorithm to previous work in section 8. Results can be found in section 7. Finally the conclusion and future work are in section 9.

2. REPLICA assembly programming model

We refer to a single chained VLIW word as a *line*. Sub-instructions on the same line are to be issued at the same time. In contrast to traditional VLIW architectures, the sub-instructions can be dependent due to chained functional units. Different types of sub-instructions are executed in different functional units, in REPLICA we distinguish between the following sub-instruction types[4]:

- Memory unit sub-instructions: Load, store and multi-prefix instructions.
- ALU sub-instructions: add and subtract etc.
- Compare unit: Compare sub-instructions which, set status register flags.
- Sequencer: branch instructions, jump etc.
- Operand: sub-instruction for loading constants, labels etc. into an operand slot
- Writeback: sub-instruction for copying register contents.

In Table 1 different configurations are shown. The simplest configuration, T5, has one ALU before the memory unit and then a compare unit and a sequencer after each other.

When programming on assembly level for the architecture, one has to distinguish between two types of register storage for intermediate results:

- *general purpose registers* `R1` to `R30` can store values persistently;
- *output buffers* `O0` to `Ox` [1], `A0` to `Ax`, `M0` to `Mx` are transient and only valid inside the line.

Both types, however, can be used in the same way:

```
ADD0 R1,R2
ADD1 O2,A0
```

Listing 1: Using registers and output buffers as operands

As shown in Listing 1, every functional unit is bound to one output buffer, denoted by the number following the instruction mnemonic. Only one dedicated functional unit type, the write-back stage, can write values to the register file.

By using the output buffers inside the super-instruction different operations can be chained and the output of one functional unit is fed as an input to another one. This saves a lot of cycles compared to a way where every intermediate result first would have to be written back to the register file. A slight drawback is that results can only be used from the left to the right. Memory operations are an exception. They cannot be chained as latency hiding would not work any longer.

```
OP0 16     ADD0 O0,R1 LD0 A0     WB1 A0     WB2 M0
OP0 2      MUL0 O0,R2 ST0 A0,R0
```

Listing 2: REPLICA assembly example [4]

The example in Listing 2 computes at line 1 an address by adding 16 to `R1` using the first ALU, `ADD0`, the result will be available in `A0` and used to load a word. The result (available in `M0`) is copied to `R2` using the writeback instruction `WB2`, at the same time `A0` is copied to `R1`. In the second line (next execution step) `R2` is multiplied with 2; the intermediate result is in `A0` and will be stored at the address contained in `R0`.

3. Dependency Graph

In both this version and the earlier version [4] of the compiler, the support of VLIW is implemented by matching a set of pre-defined combinations of sub-instructions which we call super-instructions. Of course the instruction scheduling and register allocation will not be optimal. We try to solve this problem by splitting each super-instruction into its respective sub-instructions. User-written inline assembly code is also split up. After splitting, any dependency graph between the instructions previously built is no longer valid. We therefore need to build a new one, suited to the requirements given by both the register compression (see section 4) and the ILP scheduling pass (see section 5).

A new graph is constructed by adding one instruction after another in the existing order and determining dependencies to all previous instructions. Two classes are used to store and structure the acquired information, as shown in Listing 3.

```
class MINode {
public:
  MachineBasicBlock::iterator I;
  list<MIEdge> edges;
  unsigned predecessors;
  bool scheduled;
  // [...]
};

class MIEdge {
public:
  int latency;
  MINode* to;
  // [...]
};
```

Listing 3: Definition of the `MIEdge` and `MINode` class

The first one, `class MINode`, is used to encapsulate the `MachineBasicBlock::iterator` to one instruction. Furthermore it stores, among other attributes used later on, a list of edges to other nodes which depend on this

[1]Here, `x` denotes the number of functional units of that type minus 1. For example the T7 configuration has 3 ALUs, and thus the ALU output buffers are named `A0` to `A2`. The Oi buffers are for result of OPerand instr., the Ai of ALU instr. and the Mi of memory unit instr., respectively

Name	operands	pre ALU	memory	post ALU	compare unit	sequencer unit
T5	2	1	1	0	1	1
T7	3	2	1	1	1	1
T11	7	5	1	2	1	1
T14	7	5	4	2	1	1

Table 1: list of available standard configurations

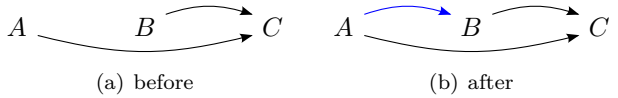
instruction. These edges are formed by instances of the `class MIEdge`.

The attribute `latency`[2] of an edge represents whether the two instructions:

- must stay within the same super-instruction, e.g. `OP0 42` $\xrightarrow{=0}$ `LD0 O0`: The operand `O0` is only valid within a line.
- the depending one *has* to be scheduled in a later super-instruction, e.g. `TRAP R0` $\xrightarrow{>0}$ `ADD0 R1,R2`: We cannot mess with the order before and after a trap.
- or it does not matter ($\xrightarrow{\geq 0}$).

The information what was previously part of one super-instruction is only important to such a degree as to determine how long transient registers are alive. Between the instruction defining the output buffer and the one using it will be an edge with latency constraint $= 0$.

In order to simplify the scheduling pass later on, additional edges are inserted between instructions that are not directly depending on each other.



(a) before (b) after

Fig. 1: Additional edges

This is only necessary for instructions that have to go into the same super-instruction. As shown in Figure 1(b), an additional edge is inserted between nodes A and B. This is required because node A would otherwise only force C into the same super-instruction and leave out B as the edges only provide a forward reference.

Compared to the dependency graph construction in the previous version of the compiler, the methods determining the dependencies have been optimized to report fewer unnecessary conflicts which results directly in fewer edges between the nodes. This in turn enables the scheduling algorithm later on to change the order of the instructions more freely and give a better result.

[2]Latency k of a data dependence edge *(i,j)* is usually defined as if $i \xrightarrow{\geq k} j$ then $t_j \geq t_i + k$ must hold for a correct schedule.

4. Register Compression

The number of slots provided for each functional unit type is limited per super-instruction. Therefore it is quite self explanatory that replacing instructions with shorter versions that do the same job. renders the resulting code more compact. As a result scheduling can pack the same code in fewer super-instructions and thereby speed up its execution. Up to now, the following substitutions are implemented:

- If the constant number 0 is required, use the register `R0` (which is there for that very purpose) instead of defining an operand with `OP 0` and then using the operand. Hence one operand slot is saved and can be used for something else. This is applicable whenever a variable is initialized or reset to zero.
- If an addition at which one operand is a constant zero (`R0`) is performed, the operation can be omitted and instead of the result the non-zero operand can be used. The same holds true for subtractions where the subtrahend is constant zero. This saves one slot in the ALU.
- With the last substitution we might end up with a situation where a register is written back to itself. Such a constellation can of course be removed entirely.
- If a constant zero (`R0`) is written back to a general purpose register, all upcoming usages of that register can be replaced with `R0` up to the next redefinition of that register.

To enforce the compiler to use register compression a flag, `-enable-replica-register-compression`, can be used. While at this point the compiler and simulator still operate under the assumption that we have unlimited write-back slots, this will not hold true in future versions. At that point saving `WB` operations will pay off.

5. ILP scheduling

5.1 Motivation

As REPLICA is a VLIW architecture, the compiler should identify operations that can be executed in parallel when there are no dependencies and resource conflicts between them, and put them in one super-instruction. Additionally the REPLICA architecture offers the possibility to chain instructions. That means that we can

use the output of one functional unit inside a super-instruction as an input to the next one. Thereby we don't have to wait until the result is written back to the register file.

When the LLVM intermediate representation is lowered to the REPLICA instruction set, the created VLIW super-instructions do represent the correct semantics, but they utilize the available functionality quite poorly. An addition operation, for example, will only make use of one ALU and one write-back slot in a super-instruction. All other slots are idling at that time. To make better usage of both the available ILP and the possibility to chain instructions, we have provided an optimization pass that reschedules the instructions, at basic block level, into new super-instructions.

Another important advantage is that this also helps us to generate code for different REPLICA configurations, i.e. providing different amounts of available slots per functional unit type.

Before we start with a description of the scheduling algorithm, some naming conventions that we will use should be introduced. The scheduler is written in a way that we support an arbitrary number of slots per functional unit type. The available slots per functional unit type as a whole will be referred to as a *bucket* (operand bucket, pre-memory-ALU bucket, memory bucket, ...). A *chain* of instructions is a list of instructions that must be scheduled in the same super-instruction. A chain is characterized by edges with latency constraint = 0. We call an instruction *ready* if all its predecessors in the dependency graph are scheduled. We call an instruction *schedulable* when it is ready and all the instructions in the chain are ready as well as there is enough space in the respective buckets to schedule the whole chain. An instruction is *emitted* when it is finally moved from its respective bucket to the output. Instructions that are scheduled but not emitted yet have a special influence during the scheduling.

5.2 Algorithm

As Figure 2 shows we start the scheduling by creating a dependency graph as described in Section 3. By implementation, this directly provides us with a list of instructions that are ready. These instructions are put into the `ready` set.

The main part of the work is split between two major steps "find next schedulable instruction" and "schedule instruction", and a third step "emit instructions" which are explained in the following.

5.2.1 Find next schedulable instruction

In this first step, we try to determine the next instruction that is schedulable. Naturally, this instruction has to be picked from the `ready` set. Hence, `isSchedulable` is called on one element (instruction) after another until we find a result. The challenge here is, that we not only have to consider the instruction itself, but the entire chain; so there has to be enough space in the respective buckets for all the instructions involved.

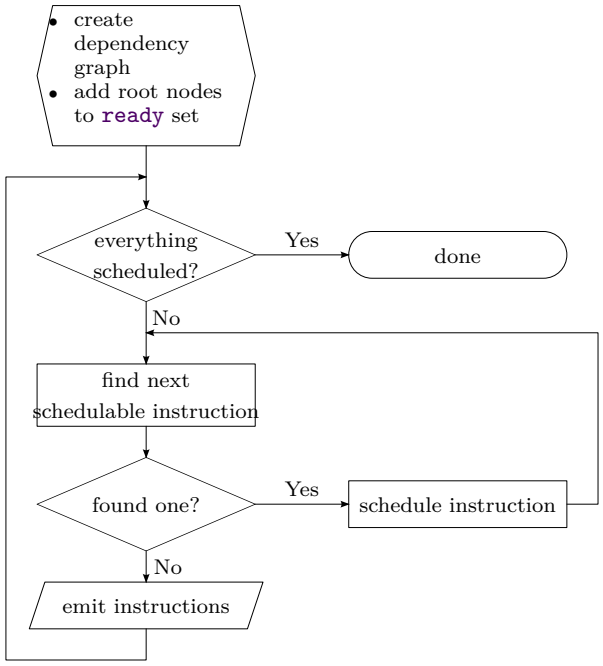


Fig. 2: Flowchart of the scheduling algorithm

A goal was to implement this in an efficient way such that we can start with an arbitrary instruction but if find at some point halfway through the chain that there is not enough space, we don't have to do a complicated roll-back of the involved data structures.

Our greedy approach does this by starting with a resource vector of type `struct RemainingSlots` that holds the amount of slots left in each bucket. This resource vector is passed by reference to recursive `isSchedulable` calls.

Every instance of `isSchedulable` now does the following steps:

- It is checked if the instruction has to be stalled because of an instruction that is already scheduled but not yet emitted. This can happen because of sequencing or memory instructions.
- The resource vector is checked if there is enough space in the respective bucket. If so, the corresponding counter is decremented by one.
- `isSchedulable` is called recursively on all depending instructions with latency = 0.

If one of these tests fails the entire attempt on the chain fails and the next ready instruction is tested.

One of the important features of the REPLICA ar-

chitecture is the chaining of instructions. Therefore `isSchedulable` checks if the current instruction is depending on a write-back (`WB`) that is scheduled but not emitted yet. If this is the case and all other prerequisites are met, we save a note that we bypass the write-back and use the output of the functional unit directly that the write-back would otherwise have to save first. Thereby we save one step. If this instruction is the only one that uses the register until it is redefined, the write-back is marked for removal altogether. At the same time we have to be aware of the order inside the super-instruction: E.g. a post-memory-ALU output cannot be used as an operand to a store instruction.

As an example, see the code in Listing 4:

```
OP0 1337        WB2 O0
OP0 42          LD0 R2,O0       WB2 M0
```

This will be rearranged to:

```
OP0 1337        OP1 42          LD0 O0,O1       WB2 M0
```

Listing 4: Skipping a writeback instruction

5.2.2 Schedule instruction

Now that we determined that an instruction plus all recursively dependent ones are schedulable, the instructions are all put in their respective buckets.

Two types of register modifications now have to be applied. Both can be observed in Listing 4.

1) The replacements that were earlier noted because we skip a writeback. `LD0` now uses `O0` instead of `R2`.
2) If we have more than one slot in a bucket, we replace the output buffer that is used. `OP0 42` was moved from slot 0 to slot 1.

All instructions that are pulled in by a latency = 0 constraint are scheduled as well. This goes fine as it was part of the `isSchedulable` check earlier.

All instructions depending on this one have now one predecessor less. Those that have reached zero predecessors are ready. Hence they are put in the `ready` set.

5.2.3 Emit instructions

If "get next schedulable instruction" can't find a suitable instruction, there are apparently not enough slots left. In this case the instructions are emitted as a VLIW and the buckets are emptied. This way of picking the next instruction will eventually cover all instructions because the initial packing into superinstructions derived by the LLVM IR sub-tree matching only imposes minimal requirements (i.e. only one slot per functional unit) and the algorithm will terminate.

6. Parametrization

In order to enforce the compiler to generate target code for a specific configuration we have introduced a set of compiler flags that we will explain briefly. Since the flags are quite many, the compiler is actually run from a script. The first parameter is `-enable-replica-ilp` which tells that the rescheduling should be applied in order to increase instruction level parallelism (ILP); for debugging purposes it can be switched off. The concept of having buckets is mentioned in the previous section, where the idea is to collect instructions that go into the same functional unit. This is intentionally built in a way that there is no restriction as to how many slots there are per bucket. The limitation of the number of slots is only imposed by passing a resource vector from an instance of `isSchedulable` to another. The initial amount (how many slots are there per bucket in a new super-instruction) is not defined in advance but rather given as a command line parameter to the compiler; `-num-ops` tells the number of integer or floating point constants available per super-instruction, `-num-alus` tells how many ALUs are available before the memory unit, `-num-mus` tells number of memory units, finally `-num-alus-post` tells the number of ALUs available after the memory access is done. All these parameters can be shown by calling `llc -help-hidden`.

As explained extensively, rescheduling tries to use the output of one functional unit as an input to another one and thereby bypass write-back operations. Furthermore, the PRAM model implementation by REPLICA requires to hide the latency to memory with different access times. Due to internal matters, this only succeeds when memory accesses are executed in parallel and not sequentially. Therefore chaining data going to and coming from memory is not possible:`-chained-mus` is therefore set to `false` by default. `-enable-replica-inline-integration` tells if an inline assembly string be split up into separate instructions and be subject to following optimization passes. The option to not split and reschedule inline assembly can be useful for debugging purposes. It is important to notice that the dependencies of inline assembly are actually taken care of. For evaluation purposes we have the parameter `-num-reg-read` and `-num-reg-write` to limit the number of general purpose registers that can be read and written respectively in each super instruction, the default is 32 (number of general purpose registers per thread).

7. Evaluation

In order to measure the improvement achieved through the different optimization passes, several test programs were written in REPLICA baseline language. Even though the focus was on instruction level parallelism

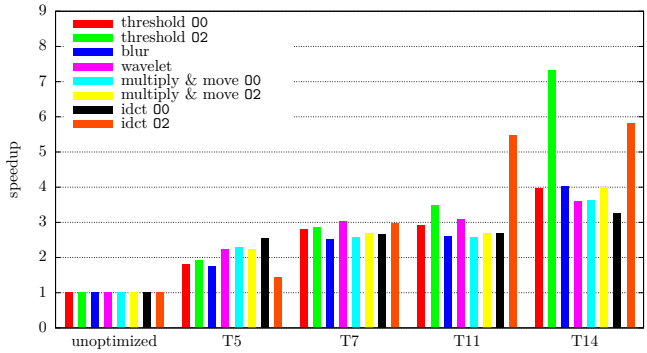


Fig. 3: Instruction level speed-up comparison with different configurations, no constraints on number of read and write ports to reg. file

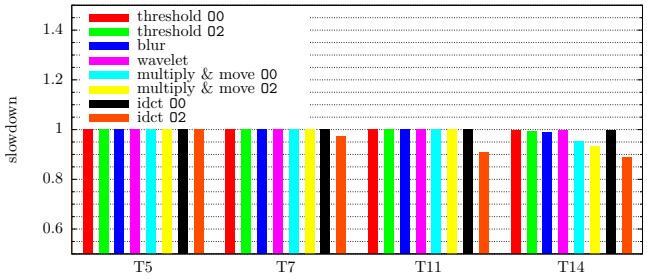


Fig. 4: Relative slowdown when limiting the number of read and write ports to four

we have also used some thread parallel programs as benchmark, to see their characteristics for instruction level parallelism. We have benchmark programs *threshold image filtering* and *blur filtering* which are both thread parallel, moreover a single threaded *discrete wavelet transform* (DWT), where our implementation is based on [5], and one simple test *multiply & move* where elements in an array are multiplied and then moved to another array. We also tested an *inverse discrete cosine transform* (IDCT), this computation kernel was extracted from the Mediabench [6] mpeg2 package.

The baseline case with unoptimized code means that no rescheduling is done, which means that unsplitted super-instructions are used directly from the mapping of the LLVM IR. For each configuration (T5, T7, T11 and T14) we did two test series for each benchmark program. One puts no constraints on the number of register read and writes, which can be seen in Figure 3. In the second test series we limited the number of both read and write ports to the register file to a maximum of four. Figure 4 displays a relative comparison with respect to performance between the two cases. It shows the expected behavior: When we increase the number of functional units without also increasing the number of available registers, speed-up will suffer. For the benchmarks multiply & move and image blur with limited register read and write ports, we will still get almost the same speed-up as in the unlimited case, this is because the functional units can be utilized well since the data dependencies allow us to make use of the buffers in the chained functional units.

8. Comparison to Previous Work

There has been a lot of work done in instruction scheduling, see e.g. [8], [9], [10], [11].

It is well-known that the time-optimal instruction scheduling problem for basic blocks is for most target architectures NP-complete [12]. While smaller scheduling problems can, today, be solved to optimality using integer linear programming and similar techniques, the general case is typically solved by heuristic algorithms.

One classical heuristic solution of the scheduling problem is Graham's greedy algorithm for scheduling tasks [13], known as "list scheduling". It maps tasks to a set of processors, where the tasks can be dependent on each other. The tasks are kept in a list ordered by their dependencies and other priorities so to assign a task to an idle processor it goes through the list and picks the next ready task. If no runnable task is found the processor will idle. Another scheduling algorithm is critical path scheduling [12].

Finlayson et al. [14] show how compiler support can help reducing power consumption without adversely affecting performance when introducing internal registers, that are read and written explicitly, instead of pipeline register. These internal registers reminds a lot of REPLICAs explicit functional unit result registers.

When it comes to VLIW scheduling for architectures with chained functional units we are only aware of the following two, which we compare in detail.

8.1 Original virtual ILP algorithm

Scheduling for a more generalized version of this type of architecture has been proposed by [7]. The algorithm proposed there proceeds in a different way than we do. It is more focused on filling the available memory slots. It sees the next free slot (both memory slots and other ones) and then tries to find a suitable instruction that is independent of all the remaining unscheduled instructions. From our point of view, this has some disadvantages:

- a lot of instructions are inspected which then turn out to have the wrong type.
- it is only taken into account that this instruction depends on other ones, not that other instructions (in this chain) might depend on this one and therefore have to fit into the same super-instruction/ VLIW.

Our algorithm, on the other hand, scans through the set of ready instructions and checks if they can be scheduled.

This seems to be more flexible if dependencies in both directions have to be considered.

8.2 Earlier ILP scheduler for the REPLICA compiler

Together with the general compiler back-end, Åkesson [4] also implemented an optimization pass that aims at exploiting more ILP. The implemented algorithm is based on [7] but looks for the next instruction (as we do) instead of the next slot. As the necessity to manage instructions in a different stage was recognized, different containers exist for holding instructions:

- those that have to be scheduled immediately (within the same super-instruction)
- those that have to be scheduled whenever there is enough space
- those that have to be scheduled at the earliest in the next super-instruction.

One can see that this corresponds to the different latency constraints defined in the dependency graph. This design, however, turns out to be rather cumbersome when we try to use short-cuts and skip write-backs. In order to do this scheduling, a dependency graph was required as well in [4]. The implementation however took in many cases a too conservative approach (e.g., it ignored the fact that output buffers are only valid inside a super-instruction) which resulted in too many unnecessary dependencies that rendered rescheduling more difficult or even impossible. [4] only support one basic REPLICA configuration.

9. Conclusion and future work

We have shown that our implementation of a parametrizable instruction scheduling algorithm for a VLIW processor with chained functional units produces high quality code for different hardware configurations. The high parametrization of the compiler makes it, together with the simulator, useful for evaluating different hardware configurations.

Future work includes the extension of scheduling beyond basic blocks, for example for loops using software pipelining techniques. It can be interesting to evaluate it for the REPLICA architecture since the case of chained functional units is a different one compared to standard VLIW architectures.

Another interesting idea would be to try to formulate and solve the scheduling problem as an integrated code generation problem of instruction selection, scheduling and register allocation together, for instance using *integer linear programming* both at basic block level and beyond. In earlier work Eriksson [15],[16] models integrated code generation for clustered VLIW DSPs using this technique. One example of similarity is that clustered VLIW DSPs have different register files which give constraints on which registers can be used by the different functional units; in our case we have the transient functional registers which are exposed to the programmer and can only be used from left to right and are only valid during one super-instruction step. Both lead to strong coupling between register allocation and instruction scheduling, for which integrated code generation provides higher code quality, see Eriksson [15].

Acknowledgment

This work was funded by VTT, Finland. The authors would like to thank Martti Forsell for his comments.

References

[1] REPLICA, project home-page, http://www.vtt.fi/sites/replica/?lang=en accessed Feb. 10. 2012

[2] J. Keller, C.W. Kessler, and J. Träff. Practical PRAM programming. Wiley series on parallel and distributed computing. J. Wiley, 2001.

[3] M. Forsell. Realizing Multioperations for Step Cached MP-SOCs. Proc. SOC'06, pages 77–82, 2006.

[4] Daniel Åkesson. An LLVM back-end for REPLICA code generation for a multi-core VLIW processor with chaining, LIU-IDA/LITH-EX-A-12/007-SE,Dept. of Computer and Information Science , Linköping University 2012.

[5] G. Pau. Fast discrete biorthogonal CDF 5/3 wavelet forward and inverse transform http://www.embl.de/~gpau/misc/dwt53.c accessed March 9. 2012.

[6] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In International Symposium on Microarchitecture, 1997.

[7] Martti J. Forsell. Using parallel slackness for extracting ILP from sequential threads, In Proc. of the SSGRR-2003s, Int. Conf. on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet, 2003, L'Aquila, Italy 2003

[8] G. Wood, Global optimization of microprograms through modular control constructs, Proc. 12th Annual Workshop in Microprogramming, 1979

[9] J.A Fisher, Trace Scheduling: A Technique for Global Microcode Compaction, IEEE Trans. on Computers, vol. 30, no. 7, 1981.

[10] M. Tokoro, E. Tamura, T. Takizuka. Optimization of microprograms, IEEE Trans. on Computers C-30:7, 1981.

[11] A. Aiken, A. Nicolau, A Development Environment for Horizontal Microcode, IEEE Trans. on Software Engineering, no. 14, 1988.

[12] C.W. Kessler. Compiling for VLIW DSPs, book chapter, 38 pages, in: S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, eds., Handbook of Signal Processing Systems, Springer, Sept. 2010

[13] R. L. Graham. Bounds for certain multiprocessing anomalies. The Bell System Technical Journal, XLV, November 1966.

[14] I. Finlayson, G. Uh, D. Whalley, G. Tyson. An Overview of Static Pipelining. In Computer Architecture Letters (CAL), accepted 2011.

[15] Mattias Eriksson. Integrated Code Generation Dissertation. Linköping Studies in Science and Technology, Dissertation No. 1375, Linköping University, Sweden, June 2011.

[16] Mattias Eriksson, Christoph Kessler. Integrated Modulo Scheduling for Clustered VLIW Architectures. Proc. HiPEAC-2009 High-Performance and Embedded Architecture and Compilers, Paphos, Cyprus, Jan. 2009. Springer LNCS 5409, pp. 65-79.

New Advances in Asynchronous Agent-based Scheduling

Jack Harris[1] **and Matthias Scheutz**[2]
[1]School of Informatics and Computing, Indiana University, Bloomington, IN 47404, USA
[2]Department of Computer Science, Tufts University, Medford, MA 02155, USA

Abstract— *Traditional discrete event simulations enumerate the event-space in a sequential manner to guarantee the consistency of the simulation. Rrecent asynchronous agent-based scheduling work has demonstrated that it is also possible to achieve consistent simulations under certain constraints even when all agents are at different time steps. This paper extends asynchronous event-based scheduling for agent-based simulation by introducing a scheduling policy based on the notion of* Dynamic Goal-based Agent Prioritization *(D-GAP) which provides the opportunity for evaluating a simulation signifcantly faster and with fewer computational steps.*

Keywords: parallel scheduling, discrete event simulation, asynchronous scheduling policies, scheduler efficiency

1. Introduction

Scheduling in traditional discrete event simulation enumerates the event-space sequentially by updating all entities at time t before evaluating any entity at the next time step t+1. This serial temporal updating limits the utility of large-scale computational resources by requiring parallel processes to synchronize at every time step in order to maintain a consistent state configuration. Scheutz and Schemerhorn defined a methodology to identify entity sets that are "update-independent" and could temporally advance in the simulation without requiring synchronization updates, thereby maximizing parallel processing for discrete event simulations [1]. Scheutz and Harris, utilizing the notion of asynchronous discrete event scheduling, published a set of policies explicitly aimed at minimizing the runtime of distributed agent-based simulations [2]. This paper builds upon that prior work. First, we introduce an important performance optimization to the previously published general asynchronous scheduling algorithm. This improvement identifies the minimum set of agent state updates required by simulations to asynchronously advance a given agent to the next time step. Secondly, we introduce a novel scheduling policy, *Dynamic Goal-based Agent Prioritization* (D-GAP). This method utilizes a directed search technique to explore a simulation's discrete event-space targeting the events that will lead to simulation completion as defined by the modeler. Together, these two advances further optimize discrete event simulations for both serial and parallel execution environments by evaluating a simulation faster and with fewer computational steps.

2. Background and Related Work

The study of discrete event simulation has been driven by the need to efficiently model complex system interactions. To facilitate the development and execution of these models, formalisms such as the Discrete Event System Specification (DEVS) have been created [3]. Parallel and distributed resources are commonly leveraged to increase the performance of model execution runtime environments for DEVS and other discrete event simulators. (See [4] for an example of an agent-based simulation modeled using DEVS methodology and executed in parallel using the High Level Architecture (HLA) distributed simulation protocol.)

Though these discrete event systems utilize parallel resources, their efficiency is reduced by the need to continuously synchronize the distributed resources and update each agent in the simulation at each time step. It is sometimes interesting (or required) to evaluate the state of every element in a simulation at simulation termination, but it is often common for a modeler to only be interested in a small set of critical simulation elements (e.g., whether one agent reaches a particular location in the environment)..

In 2006, Scheutz and Schermerhorn introduced the notion of "update-independence" for segregating groups of agents across distributed resource pools and for determining how long they could operate in isolation without sharing agent state information. Formally, an agent A_1 is update-independent from A_2 if A_1 will advance from configuration state C at time t to C' at t+1 regardless of the existence of A_2 in the simulation. Fortunately, it is possible to identify some forms of update-independency without actually advancing a simulation from t to t+1 provided that there is some a priori constraints on agent interactions (e.g., for spatial agent-based simulation: maximum velocities, sensory ranges and effector ranges of agents). For example, if an agent A_1 is outside of agent A_2's maximum interaction range and A_2 is outside of A_1's maximum sensor range then A_1 is update-independent of A_2. However, the reciprocal relation must also exist for A_2 to be independent of A_1. Both A_1 and A_2 must be mutually update-independent for either to update without causing a misconfiguration in the simulation. Clearly, it is possible that A_1 would not be affected by A_2 even if it was not outside of these ranges, but this cannot be known for certain without executing the simulation. A conservative estimation of dependence in a simulation will always ensure consistent configurations are produced.

The notion of an agent's "event-horizon" was introduced to facilitate update-independence determination. An "event-horizon" defines the spatial region containing all possible positions for that agent in a future time step. This region is determined by assuming an agent traveled at its maximum velocity from current simulation time to the projected future time in all directions simultaneously. This assumption allows "event-horizons" to be calculated using only simple geometry. Using "event-horizons", it is possible to identify agents that must be update-independent from other agents even when they exist at different simulation times.

When determining update-independence and projecting "event-horizons" of an agent, a subset of interdependent agents are identified for a given time step. This subset of agents defines a local world that could function independently from the rest of the agents in the simulation. This local world is essentially a "transitive closure" of dependency, where dependency is recursively identified for a given agent and time. Scheutz and Schemerhorn exploited this characteristic of local world independence and used it to aid the parallelization of agent-based simulations. By distributing these sets of agents across different computational resources, performance gains were achieved. These parallel units could update asynchronously until the "event-horizons" of agents inside and outside the closure intersect. When such an intersection occurs the independence of the sets is removed and the agents at the future time step become blocked and unable to continue to advance asynchronously without additional information. Therefore, to maintain a consistent simulation, the asynchronous scheduler must never run an agent asynchronously beyond the point of being blocked [1].

The formalism and correctness proofs of Scheutz and Schermerhorn provided the foundation for future agent-based asynchronous discrete event scheduler research. Subsequent research by Scheutz and Harris explored new update scheduling algorithms to optimize the discrete-event scheduler and minimize the runtime of simulations [2]. From that work novel asynchronous agent-based scheduling policies were presented; each optimizing different heuristics. For example, the $Remote - Event - First$ policy attempted to minimize simulation runtimes by minimizing the occurrences of blocked agents in a distributed context. This was accomplished by preemptively advancing agents whose projected "event-horizons" would first interact with agents on other parallel systems.

3. Improving Asynchronous Scheduling

This section outlines efficiency improvements made to the general asynchronous scheduling algorithm originally proposed by Scheutz and Harris [2]. These advancements focus on minimizing the number of dependent agent updates required to advance a given agent asynchronously in the simulation. Before the new algorithm is proposed, the original methodology is discussed as a counterpoint. Following the description of the new algorithm, a proof for correctness is presented.

Algorithm 3.1: ORIGINAL METHOD(W)

procedure ISCOMPLETE(W)
return (has terminating criterion been met)

procedure PICK(W)
comment: select and return an agent to update

procedure GETTRANSITIVECLOSUREFOR(a, t, W, S)
comment: gets a's recursively dependent set at time t
for each $d \in senseOrAffectAt(a, t, W)$ **and** $d \notin S$
do $S \leftarrow S \cup \{d\} \cup$ GETTRANSITIVECLOSUREFOR(d, t, W, S)
return (S)

procedure ISOLATEDEPENDENTSUBSET(a, W)
$Ta \leftarrow$ GETTRANSITIVECLOSUREFOR($a, time(a), W, \emptyset$)
return (Ta)

procedure UPDATESET(S)
comment: advance agents youngest to oldest
$St \leftarrow sortByLocalTime$(S)
for $i \leftarrow 0$ **to** $size(St) - 1$
do if $i < size(St) - 1$

then { **if** $time(St[i]) <= time(St[i+1]$
  then { UPDATEAGENT($St[i], S$); $i \leftarrow 0$
  else UPDATEAGENT($St[i], S$)

procedure UPDATEAGENT($a, transitiveClosureForA$)
comment: transitions a's state from time(a) to time(a)+1

main
repeat
$a \leftarrow Pick(W)$
$S \leftarrow IsolateDependentSubset(a, W)$
UPDATESET(S)
until ISCOMPLETE(W)

3.1 Full Transitive Closure Update

The original algorithm, *Full Transitive Closure Update*, ensured consistency in simulations by fully synchronizing a set of agents when a dependency relationship was identified. For example, suppose agent A_1 is at time t and agent A_2 is at time t_0 | if agent A_1's sensory range intersects with the projected "event-horizon" of agent A_2 then agent A_2 would have to advance to time t before agent A_1 could update. Furthermore, this update requirement for agent A_2 is recursively required for any agents that agent A_2 could potentially have interacted with when projected to time t. Essentially, the sum of all A_1's recursive dependencies when projected to time t would have to be updated to time t before A_1 could advance. (See Algorithm 3.1 above for the abstracted pseudocode description.)

To maintain a consistent simulation configuration, the original method updates the entire transitive closure to the same time step; however, this level of synchronization is not necessary. When utilizing a *Full Transitive*

Closure Update, notice that an agent returned in the $IsolateDependentSubset$ calculation will be updated to t+1 even if the agent's actions removes the possibility of any future interactions with other agents after a single time step. For example, if the agent travels in the opposite direction its projected "event-horizon" would look very different after just a single update.

Algorithm 3.2: NEW METHOD(W)

procedure ISCOMPLETE(W)
 return (has terminating criterion been met)

procedure PICK(W)
 comment: select and return an agent to update

procedure INCREMENTALDEPENDENTSET(a, t, W, S)
 comment: gets a's incrementally dependent set
 for each $d \in senseOrAffectAt(a, t, W)$ **and** $d \notin S$
 do $\begin{cases} i \leftarrow time(d) \\ S \leftarrow S \cup d \\ S \leftarrow S \cup \text{INCREMENTALDEPENDENTSET}(d, i, W, S) \end{cases}$
 return (S)

procedure ISOLATEDEPENDENTSUBSET(a, W)
 comment: isolate youngest dependent subset
 $D \leftarrow \text{INCREMENTALDEPENDENTSET}(a, time(a), W, \emptyset)$
 if for all $b : b \in D$ **and** $time(a) = time(b)$
 then return (D)
 $Y \leftarrow getYoungestElementsInSet(D)$
 return (Y)

procedure UPDATESET(S)
 comment: advance agents in the temporally synchronized set
 for $i \leftarrow 0$ **to** $size(S) - 1$
 do UPDATEAGENT($St[i], S$)

procedure UPDATEAGENT($a, transitiveClosureForA$)
 comment: transitions a's state from time(a) to time(a)+1

main
 repeat
 $a \leftarrow Pick(W)$
 $S \leftarrow IsolateDependentSubset(a, W)$
 UPDATESET(S)
 until ISCOMPLETE(W)

3.2 Incremental Dependency Removal

The new algorithm, *Incremental Dependency Removal*, 'loosens' the constraints on an agent's update-independence by updating only a subset of the agents identified by the original *Full Transitive Closure Update* method a single time step on each scheduler iteration. This allows for the possibility that dependencies between agents will be removed with the additional state information provided by the incremental update. Like the previous algorithm, an agent is identified to update asynchronously at the beginning of each scheduling loop; however, unlike the previous algorithm, that agent will not update unless every agent in its transitive closure is at the same time step as that agent; otherwise, only a subset of that transitive closure will run (see Algorithm 3.2).

This means that after picking an agent to advance (A), we first identify the youngest agents within agent A's $IncrementalDependentSet$. The $IncrementalDependentSet$ is generated by recursively identifying partial agent dependencies starting with agent A. This is different from at transitive closure calculation only in that the transitive closure calculates all dependencies projected to agent A's time, while $IncrementalDependentSet$ calculates the recursive dependency relative to the local time of each agent. From this $IncrementalDependentSet$, we then obtain the set of youngest dependent agents (Y). Note that Y is update-independent, since all agents in the transitive closure for any agent in Y is also contained in Y. (A proof sketch is provided below.) The agents in Y are then advanced one time step in the simulation. For agent A to advance, it would have to be selected by the scheduler's $Pick$ method until it no longer has dependencies remaining in the past and, therefore, would advance as a member of set Y.

To prove correctness of *Incremental Dependency Removal* we must show that the agents returned from $IsolateDependentSubset$ are update-independent of all other agents in the simulation. Note that $IsolateDependentSubset$ selects the youngest members (Y) returned from $IncrementalDependentSet$; therefore these members must all be at the same time step $t = time(youngest)$. Since dependency calculations were made for each of these agents in Y (because $IncrementalDependentSet$ calls $senseOrAffectAt$ for each of them and their dependents), it follows that these dependents must all be at the same time step. Now observe that $IncrementalDependentSet$ is the same algorithm as $GetTransitiveClosureFor$ when the time of each agent is the same, since each recursive call will pass in the same value (t) for the time parameter. Therefore, $IncrementalDependentSet$ contains a transitive closure for each of the youngest agents in that set and no member of these transitive closures exist at a time step other than t. Furthermore, selecting all the youngest agents at time t results in a set with no external dependencies (i.e., an update-independent set).

3.3 New Dynamic Goal-based Agent Prioritization (D-GAP) Policy

Past asynchronous scheduling policies were designed to optimize system resources and minimize simulation runtimes in parallel environments. However, often times the schedulers would pick suboptimal agents to advance asynchronously through the simulation due to the fact that they had no knowledge of the simulation goals or an agent's likelihood of achieving those goals. This issue motivated the idea of biasing the scheduler's agent selection process by

dynamically prioritizing agents based on their probability of achieving a desired goal for the simulation. While still maintaining all the constraints required for ensuring consistent simulation configurations, the *Dynamic Goal-based Agent Prioritization Policy* (D-GAP) allows agent priorities to be adjusted during a simulation execution thereby influencing which agent will advance sooner.

The D-GAP policy, like any other asynchronous scheduling policy, will not speed up every simulation. For example, if a modeler wants to know the cause of death for all agents in a simulation, the D-GAP policy would have to make the same number of updates as a traditional discrete event scheduler. However, in a distributed context it may still be beneficial to run different asynchronous policies for maximizing parallel resources and ultimately reaching simulation completion faster. However, there are many scenarios when simulations are carried out with a more refined termination condition or goal for a simulation. For example, suppose a modeler designed a simulation experiment with the intent to understand what the cause of death will be for a particular agent. The modeller would set the terminating condition for the simulation to be when that partial agent is no longer alive. For this event to occur it is quite likely that not all agents need to be updated to the same simulation cycle, but instead only the agents required for the particular event to occur. The ideal scheduling policy for this example would be to utilize the D-GAP policy.

In situations involving a subset of agents from the simulation, where a particular event or phenomenon is being studied the D-GAP policy can greatly improve a scheduler's efficiency and lead to much faster runtimes. Setting the agent of interest to a higher priority will let the scheduler identify the agents to update that will push the agent of interest through the simulation event-space fastest. This will ultimately lead to any phenomena relating to those higher priority agents to also take place sooner since less superfluous updating of other agents would occur. The agents of interest do not exist separate of their environment and other agents; therefore, other agents with lower priority would still need to be updated. These lower priority agents would update when the agents of interest progress far enough into the simulation timeline that they required additional updated state information from agents at lower priority levels.

4. Evaluation Method

The performances of the D-GAP policy and the new asynchronous scheduler optimizations were measured using a new standard metric for evaluating asynchronous agent-based schedulers. Scheduling efficiency is determined by comparing the number of agent updates conducted in the new methods with what would have been required using a traditional sequential discrete event scheduler. The number of agent updates can be calculated by incrementing a counter each time an agent's update function is called or by summing the local simulation times of each agent in the simulation. This method of comparing the entity's *update count* is preferable to simply comparing runtimes because the former efficiency metric is more general in that it is not dependent on confounding factors such as simulation-unique agent computational costs or communication delays of a particular distributed context. Therefore the evaluation of the D-GAP policy and new asynchronous scheduler updates focused on comparing the three different configurations running a simulation using: (1) a traditional sequential discrete event scheduler, (2) the D-GAP policy with the original asynchronous scheduler, and (3) the D-GAP policy with the improved asynchronous scheduler.

To evaluate the scheduling policies, an a-life simulation was constructed within the SimWorld agent-based simulation framework [12]. The SimWorld system provides an extensible framework for authoring agent-based simulations complete with an integrated reusable asynchronous scheduling system. This scheduler was modified to include the new agent-based prioritization policy as well as the ability to use the incremental dependency removal method. The Alife simulation consisted of a modified version of Scheutz, Harris and Boyd's computational agent-based model of biological model organism *Hyla versicolor* ("gray treefrog") that was originally used to identify the dominant mating strategy of these animals [13].

In the model, each male calls at a given rate and females select the closest mate that exceeds some threshold of call quality. The distribution of the agents can be observed in Figure 1. In this modified configuration the simulations were run until a particular agent of interest (i.e., male2) mated. After each scheduler iteration, the SimWorld scheduler checks if the simulation should terminate (i.e., if male2 has mated). Also at this time the agent update priorities were dynamically modified to bias the scheduler towards updating agents that would bring about the terminating condition.

5. Case Study: Results

This section contains the results of the three different run conditions discussed in the previous section. Results consist of an image for the final state of each simulation configuration as well as an efficiency metric, *update count*, presented for each simulation. Finally, an efficiency comparison is made between the three conditions.

a) Traditional Sequential Discrete Event Simulation.: The base case for this comparison was to execute the simulation using a traditional sequential discrete event system. This provides a total picture of all agent updates in the simulation, including those of interest and those that are irrelevant to the phenomenon of study. After running the initial base case of the simulation (see top image in Figure 1) we see that the simulation terminates after 49 time steps with female9 mating with the agent of interest, male2. To reach

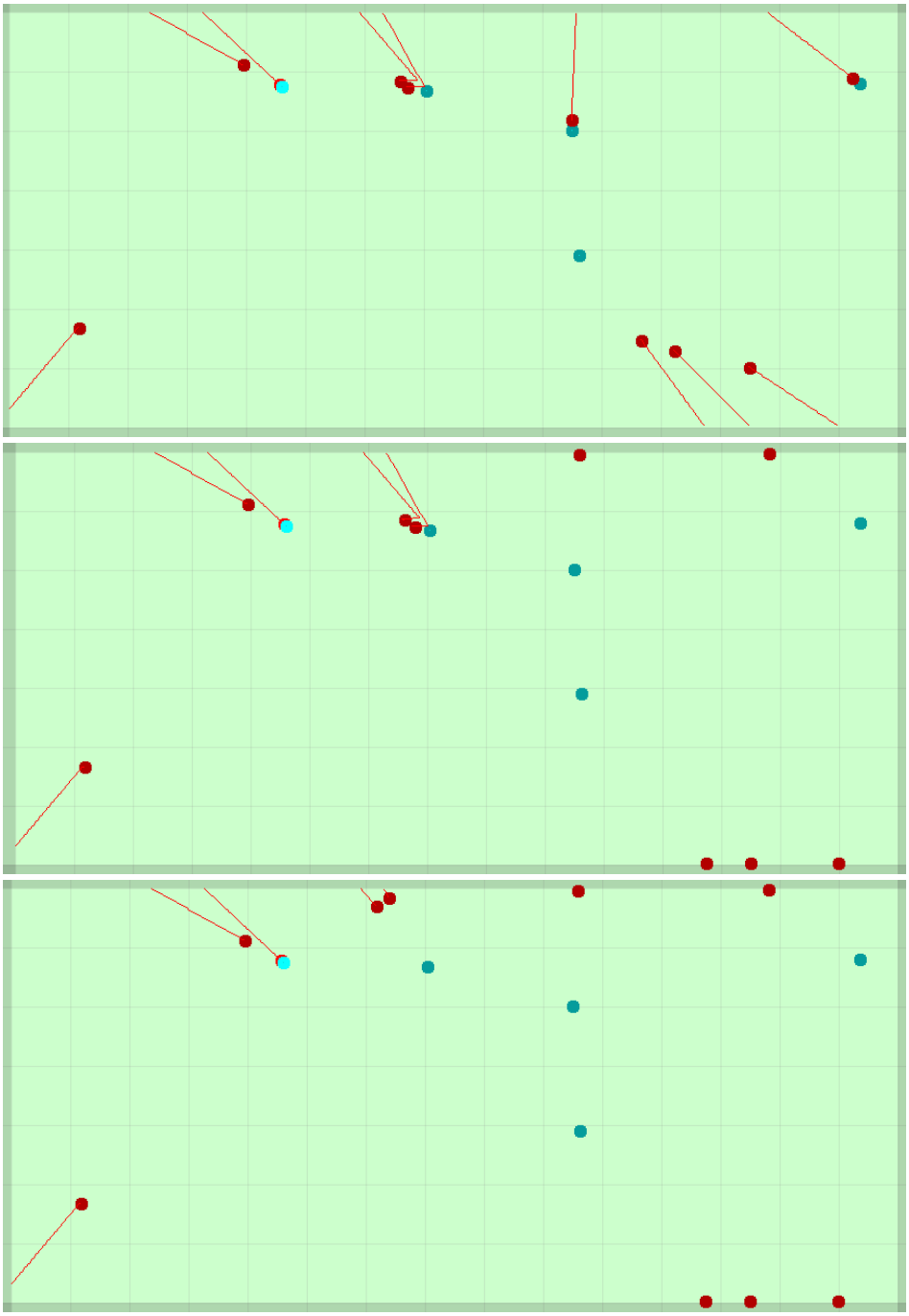


Fig. 1: *Top:* Traditional Sequential Discrete Event Simulation; the final configuration of the Alife tree-frog mating simulation. Female tree frogs (red) approach male tree frogs (blue) based on proximity and male call quality. Ultimately male2 mates with female9 (light blue) at time step 49 after executing **735** agent updates. *Middle:* D-GAP policy + Original *Full Transitive Closure Update* Asynchronous Scheduler. *Bottom:* D-GAP policy + New *Incremental Dependency Removal* Asynchronous Scheduler

the simulation's terminating condition (i.e. male2 finding a mate), **735** *agent updates* had to be calculated. From this image we see that a total of 3 females set out to attempt to mate with male2 at the beginning. By the time that the mating occured at time step 49, we can see that 2 other females had also chosen and began to pursue male2 for a

mate, however we also can notice that most of the other agents in the simulation have no bearing on male2.

b) D-GAP policy + Original 'Full Transitive Closure Update' Asynchronous Scheduler.: The simulation was then reinitialized using the exact same conditions and terminating criteria using the D-GAP policy with the originally published asynchronous scheduling algorithm. Since the simulation only updated agents in a way to ensure no inconsistent simulation states could emerge, the predictions of the simulation were identical to that of the base case; that is, male2 mated at cycle 49 with female9. However, in this case it only took **347** *agent updates* to generate this mating prediction. This algorithm, therefore, produced a scheduling efficiency gain of almost 53% for this scenario $[(735-347)/735*100]$. This a massive speed up given that the same number of processors were used in both cases and the predictive models of the simulation were not changed in anyway. When studying the middle image of Figure 1, we notice that most of the agents in the simulation did not advance far into the simulation's timeline. We will now walk through the logic by which some agents were updated and others were not. Upon simulation initialization all agents had the same priority level (unset) and all agents therefore were allowed to advance synchronously one time step. Following this initial step the *isComplete* check ran, returned false, and set the priority of the 3 leftmost females to have a higher priority since they had chosen to pursue the agent of interest Since females in this simulation do not interact with each other, these agents were *update-independent* until they could interact with a male. Therefore, the following simulation iterations allowed these three females to update asynchronously until female9 entered the interaction range of male2. Since male2 was in the past, he would need to be updated before female9 could advance and mating could take place. However, male2 could not advance independently to the female9's local time step since the "event-horizons" of two other females when extrapolated to female9's time could have potentially caused an earlier interaction with male2. Furthermore, these additional females could also have interacted with an another male when projected to female9's local time step, and, therefore, this male would also have to be included into the group that defined female9's transitive closure which would need to be updated before female9 could advance. From the simulation graphic we see that these additional female agents chose to travel in the opposite direction from male2 and essentially removed themselves from blocking female9's *update-independence*. Unfortunately the original general asynchronous scheduling algorithm still updated these agents to the female9's time. Following these updates female9 was free to proceed to mate with male2 at time step 49.

c) D-GAP policy + New Incremental Dependency Removal Asynchronous Scheduler.: Like the previous condition, the final condition utilized the D-GAP policy for selecting agents to run. However, in this case the new *Incremental Dependency Removal* optimization to the general asynchronous scheduling algorithm was also implemented resulting in the calculation of even fewer *agent updates*. Notice from the bottom image of Figure 1 that the two middle females only progressed a few time steps into the simulation's timeline as compared with the previous simulation (middle image of Figure 1). In this condition only **221** *agent updates* had to be calculated to produce the same predictions as the previous 2 cases. This results in a scheduling efficiency gain of almost 70% $[(735-221)/735*100]$.

The logical event trace for this simulation starts out similarly to the previous case. All agents update one cycle since no agent priority is initially set. Following that, the 3 females that chose to pursue the agent of interest (male2) receive higher priority in the system. The 3 female agents can all update independently until female9 becomes dependent on male2's state prior to attempting to mate. Male2 was then required to update to the same time step as female9. However, unlike the previous case, male2 was allowed to update by only one time step per scheduler cycle. At each scheduler cycle the high priority agents including female9 were selected. This again resulted in a male2 update and this continued until male2 was no longer update-independent with respect to the top middle females. The additional females were incrementally run for one step. The process continued with female9 requiring male2 to update, but in some cases male2 was able to update independently through the timeline since the top middle females actually move away from the agent of interest. The result of this incremental update strategy was that the two top middle females did not have to progress through the simulation timeline nearly as far and did not require the additional male to update at all (see bottom image of Figure 1).

5.1 Efficiency Comparison

Ultimately this case study illustrates a large performance gain from using the D-GAP algorithm and addition efficiency gain as a result of the optimization added to the general asynchronous scheduling algorithm (see Figure 2).

The D-GAP policy when coupled with the original asynchronous algorithm drastically outperforms the traditional synchronous discrete event simulator for this scenario. This improvement comes from the ability of the policy to prune unnecessary computations from the timeline based on prioritizing agents that are most likely to accomplish the research goals of the simulation. Essentially the timeline is expanded in a greedy manner such that the important events occur sooner and with less updating of agent configurations.

The additional efficiency gain achieved from using the *Incremental Dependency Removal* stems from the way this algorithm handles updating dependent agents. Instead of blindly updating agents to the point of projected possible

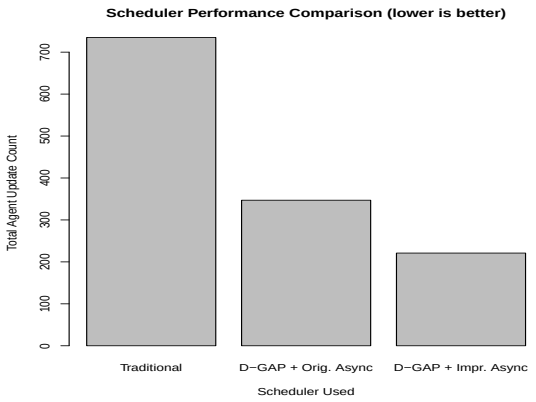


Fig. 2: Comparison of the efficiency of the 3 scheduling methods in terms of the number of *agent updates* that had to be accomplished before reaching simulation completion. The D-GAP policy with the improved asynchronous scheduling algorithm produced the most efficient runs.

interaction, dependent agents are updated only as far as necessary. Instead of updating all the agents in female9's *original* transitive closure, only the most dependent agents had to be updated incrementally. This allowed them the possibility of removing themselves from future dependencies and the need to update as much.

6. Discussion and Conclusion

The power of the D-GAP policy, like any greedy heuristic search, is only as strong as the heuristic used. In the case study, the priority assignment was clearly connected to the agent that could bring about the terminating condition. However, if priority was assigned to agents in an unrelated way to the goals of the simulation or even in an opposite way, the simulation could potentially take longer than traditional sequential scheduling. In the extreme case, it is possible for agent prioritization to be assigned such that the simulation never terminates. For example, if agent was given the highest priority in the simulation and that agent moved at maximum velocity away from the rest of the agents, it is possible that it would not interact with any other agent's projected "event-horizons" and maintain update independence indefinitely. This would lead to an agent being updated each simulation cycle that would never bring about simulation termination.

Therefore, the assignment of priority to agents must be done in an admissible way that guarantees simulation termination. There are many ways that this can be implemented. For example, adding stochasticity to the priority assignment will guarantee simulation termination even in the worst-case scenario when agent prioritization is completely assigned backwards. Another method would be to progressively penalize agents too far in the future so that there is a maximum time gap between the oldest and youngest agents in the simulation. The latter provides a bounded level of asynchrony and greedy search while maintaining the assurance of simulation termination.

Asynchronous discrete event scheduling is an extremely new concept. There have only been a handful of algorithms and policies developed to exploit the advantages of asynchronous scheduling. Policies developed so far have either optimized the characteristics of the distributed environment in which they were implemented (e.g. $Remote - Event - First$ or $Youngest - Unblocked - First$ [2]) or have attempted to optimize the scheduling using some heuristic to bias local scheduling of agents (e.g., the D-GAP policy described in this paper). Interesting future work would include policies that consider both the goals of the simulation and the distributed environment in which they are executed when making decisions on which agent to asynchronously guide through the simulation space first.

References

[1] M. Scheutz and P. Schermerhorn, "Adaptive algorithms for the dynamic distribution and parallel execution of agent-based models," *Journal of Parallel and Distributed Computing*, vol. 66, no. 8, pp. 1037–1051, 2006.

[2] M. Scheutz and J. Harris, "Adaptive scheduling algorithms for the dynamic distribution and parallel execution of spatial agent-based models," in *Parallel and Distributed Computational Intelligence*, ser. Studies in Computational Intelligence, F. Fernández de Vega and E. Cantú-Paz, Eds. Springer, 2010, vol. 269, pp. 207–233.

[3] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation, Second Edition*, 2nd ed. Academic Press, Jan. 2000.

[4] M. Lees, B. Logan, and G. Theodoropoulos, "Distributed simulation of agent-based systems with HLA," *ACM Transactions on Modeling and Computer Simulation*, vol. 17, no. 3, p. 11, 2007.

[5] M. B. Ptacek, "Calling sites used by male gray treefrogs, Hyla versicolor and Hyla chrysoscelis, in sympatry and allopatry in Missouri," *Herpetologica*, vol. 48, no. 4, pp. 373–382, 1992.

[6] G. M. Fellers, "Aggression, territoriality, and mating-behavior in North-American treefrogs," *Animal Behaviour*, vol. 27, no. FEB, pp. 107–119, 1979.

[7] ——, "Mate selection in the gray treefrog, Hyla-versicolor," *Copeia*, no. 2, pp. 286–290, 1979.

[8] M. E. Ritke and R. D. Semlitsch, "Mating-behavior and determinants of male mating success in the gray treefrog, Hyla-chrysoscelis," *Canadian Journal of Zoology-Revue Canadienne De Zoologie*, vol. 69, no. 1, pp. 246–250, 1991.

[9] O. M. Beckers and J. Schul, "Phonotaxis in Hyla versicolor (Anura, Hylidae): the effect of absolute call amplitude," *Journal of Comparative Physiology a – Neuroethology Sensory Neural and Behavioral Physiology*, vol. 190, no. 11, pp. 869–876, 2004.

[10] S. L. Bush, H. C. Gerhardt, and J. Schul, "Pattern recognition and call preferences in treefrogs (Anura : Hylidae): a quantitative analysis using a no-choice paradigm," *Animal Behaviour*, vol. 63, pp. 7–14, 2002.

[11] H. C. Gerhardt, S. D. Tanner, C. M. Corrigan, and H. C. Walton, "Female preference functions based on call duration in the gray tree frog (Hyla versicolor)," *Behavioral Ecology*, vol. 11, no. 6, pp. 663–669, 2000.

[12] M. Scheutz, P. Schermerhorn, R. Connaughton, and A. Dingler, "Swages–an extendable parallel grid experimentation system for large-scale agent-based alife simulations," in *Proceedings of Artificial Life X*, June 2006, pp. 412–418.

[13] M. Scheutz, J. Harris, and S. Boyd, "How to pick the right one: Investigating tradeoffs among female mate choice strategies in treefrogs," in *Proceedings of the Simulation of Adaptive Behavior 2010*, 2010, pp. 618–627.

A Multi-criteria Class-based Job Scheduler for Large Computing Farms

R. Baraglia[1], **P. Dazzi**[1], **and R. Ferrini**[1]
[1]ISTI "A. Faedo", CNR, Pisa, Italy

Abstract— *In this paper we propose a new multi-criteria class-based job scheduler able to dynamically schedule a stream of batch jobs on large-scale computing farms. It is driven by several configuration parameters allowing the scheduler customization with respect to the goals of an installation. The proposed scheduling policies allow to maximize the resource usage and to guarantee the applications QoS requirements. The proposed solution has been evaluated by simulations using different streams of synthetically generated jobs. To analyze the quality of our solution we propose a new methodology to estimate whether at a given time the resources in the system are really sufficient to meet the service level requested by the submitted jobs. Moreover, the proposed solution was also evaluated comparing it with the Backfilling and Flexible backfilling algorithms. Our scheduler demonstrated to be able to carry out good scheduling choices.*

Keywords: Job scheduling; Deadline Scheduling; Software License Scheduling; Computing Farm.

1. Introduction

In large computing farms providing utility computing for a large number of users, with different functional and non-functional requirements, a scheduler plays a basic role in order to efficiently and effectively schedule submitted jobs on the available resources. The objective of the scheduling is to assign tasks to specific resources maximizing the overall resource utilization and guaranteeing the QoS required by applications. The scheduling problem has shown to be NP-complete in its general as well as in some restricted forms [3]. The scheduling on utility computing environments is multi-criteria in nature [8]. In fact, these environments generally manage computational requests dynamically time varying, and with different computational requirements and constraints that compete to access shared resources. Even if in the past research efforts has been devoted to develop multi-criteria job scheduling algorithms [7], [6], [1], [2], there is still the need to improve the scheduling techniques able to manage an increasing number of jobs and to address all the application and installation requirements as well as sets of constraints for the afore mentioned computational environment. In this paper, we propose a scheduler able to schedule a continuos stream of batch jobs on large-scale computing farms. As typical scenario we consider a computing farm made up of heterogeneous, single-processor or SMP machines, linked by a low-latency, high-bandwidth network. Some characteristics of the computing nodes (e.g. processor type, memory size, number of CPUs, link bandwidth) are static and known whereas some others are dynamic (e.g. floating sw licenses). The adopted scheduling policies permit us to optimize the scheduling with respect to different objectives, even contrasting, such as maximize the resource usage and to guarantee the non-functional applications requirements. The rest of this paper is organized as follows. Section 2 describes some of the most common job scheduling algorithms. Section 3 gives a description of the problem. Section 4 describes our solution. Section 5 outlines and evaluates our job scheduler. Finally, conclusions and future work are described in Section 6.

2. Related work

Batch jobs scheduling are mainly divided in two main classes: on-line and offline. On-line algorithms are those that do not have any knowledge about the whole input job stream. They take decisions for each arriving job without knowing future inputs. Conversely, offline algorithms know all the jobs before taking scheduling decisions. Many of these algorithms are exploited into commercial and open source job schedulers [4]. The Backfilling algorithm [9] is a widely adopted scheduling approach, it is an optimization of the FCFS algorithm [10]. It requires each job specifies its execution time, so that the scheduler can estimate when jobs finish and other ones can be started. The main goal of Backfilling is to exploit a resource reservation approach to improve the FCFS policy by increasing the system resource usage and by decreasing the average job waiting time in the scheduler's queue. In order to improve performance, some backfilling variants, such as Flexible backfilling [5] have been proposed. The Flexible backfilling algorithm is obtained by exploiting a different order of queued jobs. Jobs prioritized according to scheduler goals are queued according to their priority value, and selected for scheduling. Even if the multi-criteria approach seems to be the most viable one to solve the resource management and scheduling problem in heterogeneous and distributed computational environments, only a few research efforts have been done in such direction [1], [7], [2], [11]. In [1] a multi-criteria job scheduler for scheduling a continuous stream of batch jobs on large scale computing

farms is proposed. It exploits a set of heuristics that drive the scheduler in taking decisions. Each heuristics manages a specific constraint, and contributes to compute the measurement of the matching degree between a job and a machine. The scheduler allows its extensions to manage a wide set of requirements and constraints. In [7] K. Kurowski *et al.* propose a two-level hierarchy multi-criteria scheduling approach for Grid environments. All participants of a scheduling process, i.e. endusers, Grid administrators and resource providers, express their requirements and preferences by using two sets of parameters: hard constraints and soft constraints. A Grid broker at higher level exploits the hard constraints to compute a set of feasible solutions, which can be optimized by using soft constraints describing preferences regarding multiple criteria, such as various performance factors, QoS-based parameters, and characteristics of local schedulers. In [2] a bi-criteria algorithm for scheduling moldable jobs on cluster computing platforms is proposed. It exploits two pre-existing algorithms to simultaneously optimize two criteria: job makespan and weighted minimal average completion time. Such criteria are complementarity, and well represent the objectives of both users and system administrators. The algorithm was evaluated by simulations using two different synthetic workloads. In [11] a solution based on advanced resource reservation that optimizes resource utilization and user QoS constraints for Grid environments is proposed. It supports advanced reservations to deal with the dynamic of Grids and provides a solution for agreement enforcement. The proposed advanced reservation solution is structured according to a 3-layered negotiation protocol. Preferences of end-users are taken into account to start a negotiation to select resources to reserve. The user can select the best suitable offer or can decide to re-negotiate by changing some of the constraints. End-users preferences are modeled as utility functions for which end users have to specify required values and negotiation levels. In [6] is proposed a schedule-based solution for scheduling a continuous stream of batch jobs on computational Grids. The solution is based on Earliest Deadline First (EG-EDF) rule and Tabu search technique. The EG-EDF rule incrementally builds the schedule for all jobs by applying technique which fills earliest existing gaps in the schedule with newly arriving jobs. If no gap for a coming job is available EG-EDF rule uses Earliest Deadline First (EDF) strategy for including a new job into the existing schedule. The schedule is then optimized by using a Tabu search algorithm to move jobs into earliest gaps. Scheduling choices are taken to meet the QoS requested by the submitted jobs, and to optimize the hardware resource usage.

3. Problem Description

We consider jobs and machines annotated with information describing their requirements and features, respectively. Jobs in a stream can be sequential or multi-thread, and all the jobs are independent one from each other. To each job is attached a description containing both an identifier and a set of functional and non-functional requirements. Functional requirements include the number of processors, the RAM size and the software licenses a job needs to be executed. Non-functional requirements (also referred as QoS) are job slowdown equal to one, job deadline and job advanced resource reservation. The description also includes an estimation of the time required to compute the job and the features describing the processor exploited to perform such estimation (benchmark score). Each job is executed on a single machine, and all jobs are preemptable. Job preemption can be performed when either a job submission or a job ending event takes place. The machines composing the farm are described by a benchmark score, the number and type of CPUs, the size of the RAM installed and the non-floating (i.e. bound to a machine) and floating (i.e. not bound to any specific machine) software licenses they can run. Processors installed on each machine has associated a weight. Every machine can execute multiple jobs at the same time in a space-sharing fashion. All the machines support two basic forms of job preemption: stop/restart and suspend/resume. The checkpoint/restart form is possible only if the running job is properly instrumented. Machines are assigned to jobs in the shape of sub-machines, namely a subset of a machine's processors. A submachine is managed by the scheduler as an instance of the machine from which it is originated. Floating sw licenses can be assigned to any machine able to run them. The only limit is that the total number of licenses in use can not be greater than their availability. In our study, we consider the association of licenses to machines. As a consequence if a set of jobs requiring the same license can be executed on the same machine, only one license copy is accounted.

4. The scheduler achitecture

The proposed scheduler is based on multiple job classes. Each job is assigned to a class on the basis of its functional and/or non-functional requirements. Figure 1 depicts the architecture of our scheduler. Three main components are represented: *Job-Dispatcher*, *Class-Scheduler* and *Control-Scheduler*. The *Job-Dispatcher* receives, classifies, and dispatches each job to the proper class. A class is an entity characterized by a set of dynamically assigned computational resources, a job queue and a Class Scheduler. The classes are ranked according to a priority value assigned statically by the installation on the basis of the functional and non-functional requirements managed. To each class is associated a *Class-Scheduler* (CLS). This component is specialized for managing a specific class of job requirements. To each CLS is associated a job queue and a set of resources. The CLS extracts jobs from its queue and allocates them resources to be run. In case of resource shortage, it issues a request for additional resources to the Control Scheduler. A class releases the assigned resources when they have not been used

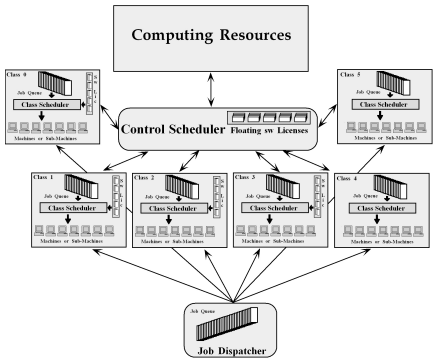


Fig. 1: The scheduler architecture.

for a predefined quantum of time, fixed by the installation. The *Control-Scheduler* (CNS) is devoted to manage requests issued by CLSs. CNS allocates resources to CLSs in the form of sub-machines or floating sw licenses. As an example, consider a job asking for four processors in a computing farm composed by only eight-processors machines. The CNS assigns four free processors from an available suitable machine to the requesting class. That class becomes the temporary owner of the assigned resources until it will not release them. Requests issued by higher ranked classes are scheduled first than the other ones, and requests issued by the same CLS are managed according to the FCFS order. CNS defines new sub-machines according to two alternatives: 1) The sub-machine is defined on a machine already assigned to the requesting class, 2) The sub-machine is defined on a machine not assigned to any class, i.e. a free machine. If no machine is available, the CNS can decide to enact a resource stealing process. The definition of a new sub-machine is performed by exploiting the principle of the least privilege, namely, from all the available machines is chosen the one with the least amount of resources that is sufficient to satisfy the requested assignment. Sub-machines are managed using a data structure consisting in a vector VFM which length equals to the number of processors P of the largest machine in the computing farm. Each element in the vector contains a list in which each elements represents a farm's machine. A machine belongs to list at index i if its actual availability of processors equals to i. The lists are arranged in increasing order with respect to machine memory size, and then the number of floating sw licenses the machine can run. In order to find a machine with p processors, a RAM of size r and l licenses, CNS starts its search from the $VFM[p]$ element and continues until it finds a machine addressing the assignment requirements or the vector ends. When a proper machine is found and a new sub-machine is created the number of available processors in that machine decreases correspondently. The data structure is updated consequently. The idea behind this data structure is to keep lager machines available for subsequent requests and to reduce machines fragmentation. Floating sw licenses are managed using a specific data structure consisting in a vector which length equals to the number of available floating sw licenses S. Each vector entry addresses a list storing the number of currently usable copies for a specific license. Such lists are structured according to three sublists storing respectively the number of available copies, the number of copies assigned to a class but not used, and the number of copies assigned to a class and in use. Floating licenses belonging to the first two sublists are available for assignment to classes, whereas the copies in the third list are already assigned. When a free copy of a floating sw license is assigned to a class, it is removed from the first sublist and assigned to the third sublist. When a job using a floating sw license finishes its execution the license copy is moved to the second sublist. After an installation-defined quantum of time licenses in the second sublist are released and moved to the first sublist. In our study we considered six job classes ($0 \div 5$) The first three classes (0, 1 and 2) manage jobs with both functional and non-functional requirements whereas the other three ones (3, 4 and 5) manage jobs with only functional requirements. Jobs are assigned to the classes according to the following criteria:

- *Class 0*: Jobs requiring a slowdown equal to 1. Jobs are managed by the related CLS according to the First Come First Served (FCFS) order.
- *Class 1*: Jobs requiring advanced resource reservation. Jobs are scheduled according their closeness to the reservation. Jobs for which the resource reservation fails are discarded. Alternatively, but not in our study, they could be moved to Class 3, 4 or 5 depending on their functional requirements.
- *Class 2*: Jobs with deadline. Jobs are scheduled according to the expected time they have to start to meet their deadline. The queue position of a job is determined exploiting the solution proposed in [6]. According to such solution, the closer the deadline of a job is, the higher its position in the job queue is.
- *Class 3*: Sequential or parallel jobs requiring floating sw licenses.
- *Class 4*: Parallel jobs not requiring any floating sw license.
- *Class 5*: Sequential jobs not asking for floating sw licenses.

Jobs within classes 3, 4 and 5 are selected by the related CLS according to the FCFS order. If a job has requirements to be assigned to two different classes, it will be assigned to the class having a higher priority. The assignment of resources to classes permits to exploit the locality in job requirements. In fact, after a initial time, it is highly probable that a class managing jobs with similar requirements owns in advance the resources to run the jobs to it assigned.

Resource stealing: When a class is experiencing a lack of free resources to satisfy a request its CLS issues a request to CNS. As a consequence, the execution of jobs belonging to classes with a lower priority have to be interrupted to release the needed resources. However, the interruption of a job has a cost for the computing farm. Actually, interrupting a job is convenient only if the gain resulting from the use of the released resources overcomes this cost. To evaluate this cost several parameters should be considered, e.g. the time elapsed in execution by the job candidate to be interrupted, the number of sw assigned to that job, etc. In our model a resource r can be moved from a class A to a class B if the following expression is verified: $Rank^A > Cost^B(r)$, where $Rank^A$ is the rank value of Class A and $Cost^B(r)$ is the cost associated to the interruption of jobs running on resource r and belongs to B. Considering a generic class C, a resource r, and the number k of jobs running on r such cost can be computed as: $Cost^C(r) = Rank^C + \sum_{i=1}^{k} P^C(i)$, where $P^C(i) = W_i \cdot (\frac{T_{ex}(i)}{T_{tot}(i)}) \cdot W_{dead} + (W_{pr} \cdot Pr(i)) + (W_l \cdot L(i))$. $T_{ex}(i)$ is the time spent executing the job i and $T_{tot}(i)$ is the total estimated execution time of the job i. W_i is the weight associated to the form of preemption adopted to interrupt the execution of the job i. This is small if the job supports checkpoint/restart, it increases in case of suspend/resume and it is maximum if the only option is stop/restart. W_{dead} is the weight associated to jobs having a deadline. It equals to 1 for jobs without deadline. W_{pr} is the weight associated to a processor and $Pr(i)$ is the number of processors assigned to i. W_l is the weight associated to a floating sw license and $L(i)$ is the number of floating sw licenses used by i. The idea of this approach is to allow an installation to tune W_i, W_{pr}, W_l and W_{dead} values and class ranks according to its objectives. As an example, suppose that an installation goal is to respect in a very strict way the prioritization given by the jobs classes.To this end, the *Rank* associated to two consecutive classes have to differ by a value greater than the maximum value that $P^C(i)$ can assume. Such value is obtained when a job i (it makes no difference to have just one or several jobs if the overall resource usage is the same) is using all the processors of the largest farm's machine, all the available floating sw licenses, and is approaching the end of its execution. Let's assume that the largest farm's machine has 1024 processors, and that the total number of floating sw licenses is 20. Moreover, consider weights assuming these values: $W_i = 200$, $W_{pr} = 1$, $W_l = 0.5$ and $W_{dead} = 2$. Hence, the maximum value $P^C(i)$ can assume is: $P^C_{max} = 200 * 1 * 2 + 1 * 1024 + 0.5 * 20 = 1434$. As a consequence, the rank values of the six classes have to be fixed as follows: $Rank\_Class5 = 0$, $Rank\_Class4 = 1500$, $Rank\_Class3 = 3000$, $Rank\_Class2 = 4500$, $Rank\_Class1 = 6000$, and $Rank\_Class0 = 7500$[1].

[1]These rank values are the ones exploited in the conducted tests

Resource search: Considering a job i belonging to a class A and requiring $Pr_x \leq P$ processors and $L_x \subseteq S$ floating sw licenses the resource search algorithm is structured according to the following steps:

1) Starting from the entry Pr_x, the VFM data structure is analyzed to find jobs to be interrupted.
2) For every machine m suitable for executing i indexed by Pr_x, a list of jobs that could be interrupted is created. The list also includes free processors.
3) The first N_x jobs, which interruption permits us to obtain the required Pr_x processors are selected.
4) If selecting the first N_x jobs a number of processors greater than Pr_x is obtained, a refined step is conducted to fix the number of selected processors. The list of selected jobs is visited in reverse order to remove exceeding processors.
5) The cost $Cost_r$ is computed as the sum of the costs related to the jobs being interrupted on m. If $Cost_r$ is smaller than the costs computed for the other analyzed machines, the machine m is selected, and the jobs on it executing are selected to be interrupted.
6) Steps from 2 to 5 are repeated from $Pr_x + 1$ to P to find machines suitable for executing i.
7) At the end of step 6, the list of jobs that cloud be interrupted (i.e. jobs running on the machines with associated the lowest costs) is carried out.

The interruption of the selected jobs may lead to free some required licenses. In this case, the found licenses are removed from L_x and the following steps are executed:

1) The floating sw licenses search starts from the queue $l \in L_x$ of the floating sw license data structure.
2) The cost $Cost_r$ due to the interruption of a job using l is computed. The job corresponding to the smallest $Cost_r$ is selected and its execution interrupted.
3) Steps 1 and 2 are repeated until all the licenses needed to run the job i are found.
4) At the end of step 3, the set of jobs to interrupt is found.

This phase is the most computation expensive one. In fact, the search of free processors requires in the worst case to analyze all the available machines and floating sw licenses. The search of processors needs to sort the N jobs running on each of the M machines in the farm. Since the sort operation has complexity $NlogN$, in the worst case, the resource search algorithm has complexity $C = M \cdot NlogN$. To search a floating sw licenses in the worst case has complexity $|L|$.

5. Performance Evaluation

The evaluation of the proposed scheduler was conducted by simulations using different streams of jobs and farms of different size. Job and machine parameters have been randomly generated from a uniform distribution in the ranges shown in Table 1. Moreover, we compared our solution

with Backfilling and Flexible backfilling algorithms. The job priorities of the Flexible backfilling algorithm are updated at each job submission or ending event and the reservation for the first queued job is maintained through events.

Table 1: Parameters used to generate jobs and machines.

Description	Range
Processor Type	$1 \div 5$
Number of processors	$1 \div 128$
Benchmark score	$0.5 \div 2$
RAM	$500Mb \div 5Gb$
Job estimated execution time (secs)	$16000 \div 20000$
Number of licenses copies	$50 \div 70$
Number of different licenses	20

For each simulations the percentages of jobs requiring specific functional and non-functional requirements have been generated according to the values shown in Table 2.

Table 2: Percentages used to generate the job steams.

Percentage	Description
5%	requires a slowdown equal to 1
30%	has a deadline
5%	needs of advanced resource reservation
60%	needs a software license
30%	needs a floating software license
10%	needs a specific hardware
2%	supports checkpointing
40%	needs 1 processor
40%	needs 2 processors
10%	needs 4 processors
0.8%	needs 8 processors
0.08%	needs 16 processors
0.06%	needs 32 processors
0.04%	needs 64 processors
0.02%	needs 128 processors

The duration of each simulation was set at 43200 time units (i.e. the number of seconds in 12 hours). For each simulation unit the system: (1) Generate a job and put it in the Dispatcher's job queue, (2) Update of the running jobs status (3) Update of the status of the resources (4) Execute the CLSs, (5) Execute the CNS, (6) Store the simulation statistics. In the conducted experiments the number of generated machines varied from 1000 to 1200, and to obtain stable values each simulation was repeated 50 times with different farm configurations and job streams. The performance metrics have been evaluated versus the system contention. Usually, this value is roughly computed as: $ResourceR/ResourceA$, where $ResourceR$ is the amount of a specific resource requested by the jobs in the system, and $ResourcesA$ is the available amount of such resource. This ratio does not provide an accurate information on resource availability because it ignores the jobs allocation implied constraints. In fact, all the requirements of a job must be satisfied to allocate it, so the variables describing the available resources cannot be considered independently. To clarify this point, let us suppose that the value computed by using the above expression is less than 1. In principle, it indicates an availability of the considered resource. As a consequence, a scheduler should be able to properly allocate the jobs on the available resource. However, this is not always true. As an example consider an availability of 20 processors in the system and a job to schedule requiring 16 processors. Clearly, if at least 16 of the 20 free processors are not available on the same machine the job can not be scheduled even if a rough analysis would suggest enough processors availability. Unfortunately, in general can be hard to understand if the resource shortage is caused by the ineffectiveness of the adopted scheduler or by an insufficient number of available resources. To overcome this problem we introduce the RRI index. Its aim is to exploit a simple allocator to measure, with a certain degree of approximation, if at a certain time, the resources in the system are sufficient to meet all the jobs requirements. In particular, in this paper we only consider the resource processor for computing the RRI index. To this end, we considered the following four job scheduling algorithms (but in principle others can be also considered), each one basing its strategy on a different job allocation policy: 1) *Largest Machine*, which allocates a job on the machine with the largest number of free processors, 2) *Smallest Machine*, which allocates a job on the machine with the smallest number of free processors, 3) *Smallest Residue*, which allocates a job on the machine where remains, after the allocation of a job, the lowest number of free processors, 4) *Largest Residue*, which allocates a job on the machine where remains, after the allocation of a job, the largest number of free processors. These algorithms were evaluated to find the one leading to the best processor usage in the simulated environment. To this end, a workload able to use all the available processors of the simulated farm was designed according to the following four steps: 1) A random generation of a set of machines, 2) For each machine a proper set of jobs were generated, 3) A random distribution of all the processors belonging to each machine to the generated set of jobs, 4) Assignment of the generated jobs to a free computation slot in such a way that they finish their execution on the target machine all at a fixed time. RRI is computed as: $(Processors_r + Processors_q)/Processors_a$, where $Processors_r$ are the processors request by the allocated jobs, $Processors_q$ are the processors request by the not allocated jobs, and $Processors_a$ are the available processors. The higher the RRI value is, the higher the system contention is. Smallest Residue is the method that obtained the best results in 500 simulations we conducting varying the number and the type of the machines inside the simulated farm. This is the allocator we used for computing the RRI index. It behaves as a sort of probe to measure the processors availability throughout a simulation. It is executed each time a job execution is started. To evaluate the scheduler efficiency, we analyzed the algorithms exploited by CNS

to handle sub-machines. Figure 2 shows the percentages of new sub-machines definition and expansion we obtained by the simulations. When the RRI value is low ($RRI < 0.4$) there is a high rate of new sub-machines definition because the classes have only a few resources assigned. When the value of RRI is between 0.4 and 0.8 there is a higher sub-machines expansion rate because the classes already have a large number of sub-machines and at the same time there are enough available processors on the farm machines. When the value of RRI is greater than 0.8 the percentages of the expanding and new definition processes tend to stabilize at values of 30% and 40%, respectively. This result means that, when the system is heavily loaded, for example, when RRI is equal to 2 the classes in about 35% of cases already have a submachine able to execute a submitted job. While, in the other 65% of cases, CLSs require to CNS to extend a sub-machine (25%) or to define a new sub-machine (40%).

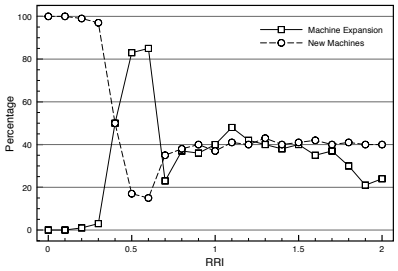


Fig. 2: New sub-machines definition or expansion.

The graph of Figure 3 shows the percentage of processors used by the running jobs. When the RRI index is greater than 1, i.e. when the requested resources begin to be unavailable, the processor utilization approaches to 100%. But the shape of the curve clearly shows that in some cases there are free processors even if the values of RRI are greater than 1. In fact, also when the value of RRI is greater than 1.2 (i.e. when the estimated number of requested resources go over in the available ones) the figure shows that some processors are not used. This happens because, also when the number of requests is much greater than the available resources, the last are not able to run any one of the waiting requests. However, it is worth to point out that our scheduler is able to schedule jobs in a way that keeps low the number of unused resources.

We also investigated the degree of satisfaction of non-functional job requirements. We evaluated the *quality of service* provided by the Control-Scheduler on the basis of decisions it has made to allocate resources to the classes. This analysis has been conducted to assess the choices made in the following areas: (1) Job classification policies and rank values assigned to classes; (2) Resource stealing technique. Bad choices can cause long queuing times for some types of jobs, in particular the ones belonging to low ranked classes.

To evaluate the satisfaction level of the resource demands

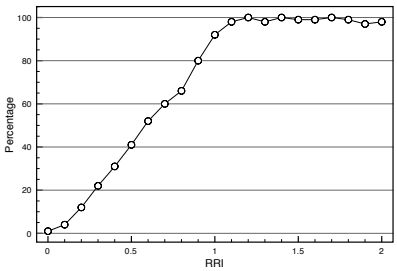
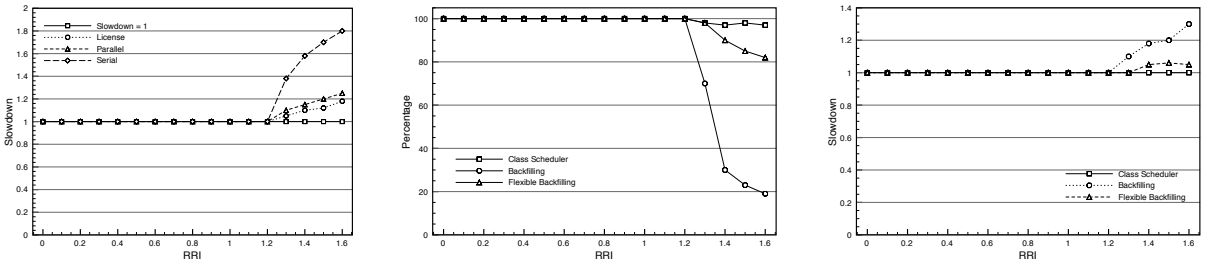


Fig. 3: Processor usage.

we used the *slowdown* metric. This measures the ratio between the response time of a job (i.e. the time elapsed between its submission and its termination) and its execution time. It is computed as: $(Tw + Te)/Te$, with Tw the time that a job spends waiting to start and/or restart its execution, and Te the job execution time [9]. Figure 4(a) shows the average slowdown obtained executing job belonging to the following job classes: Class 0, i.e. jobs requiring a slowdown equal to 1; Class 3, i.e sequential or parallel jobs requiring floating license, Class 4, i.e. parallel jobs do not requiring floating license, and Class 5, i.e. sequential jobs do not asking for floating license. In this evaluation jobs requesting advanced reservation or a deadLine were not considered because for such jobs the slowdown is not indicative. In fact, depending on their characteristics, such jobs can spend some time enqueued before to be executed without affecting their performances. In our tests all the requests of advanced resource reservation were satisfied. Figure 4(a) shows that, when the resources are available (i.e. $RRI < 1$) all the jobs obtain a slowdown equal to 1. When $RRI > 1$, i.e. the job competition to access the available computational resources increases, jobs are forced to spend some time in the queues resulting in an increase of the job average slowdown. It can be seen that in the conducted tests the job requirement *slowdown=1* is satisfied also when the value of RRI reaches 1.6 (i.e. high system contention). The slowdown value of jobs asking floating software licenses remain under 1.2 also with high system contention, while the slowdown increases only up to a 20% for parallel jobs. Instead, the value of the slowdown of the serial jobs is the worst, their completion time can increase up to 80%. Figure 4(b) shows the percentage of jobs executed respecting their deadline. The results obtained by the proposed scheduler were compared with those obtained by running a Backfilling and Flexible backfilling algorithms with the same simulation conditions, i.e. with both the same machines and the same job streams, used to evaluate the proposed scheduler. As expected, the lower the system contention is ($RRI \leq 1$), the higher the percentage of the jobs meeting their deadline is, and all the schedulers are able to satisfy all deadline requests. The proposed scheduler is able to obtain better results than the other algorithms. It obtains a percentage of



(a) Job slowdown (b) Job deadline (c) Slowdown of jobs requiring slowdown = 1

Fig. 4: Slowdown

jobs that respect their deadline very close to 100% also with a high system contention ($RRI \geq 1.5$). As the system contention increases the Flexible backfilling algorithm reaches a performance that is 16% lower than the one obtained by the proposed scheduler, while the Backfilling algorithm obtains a performance significantly lower than the one obtained by our scheduler. In Figure 4(c) we show the results obtained from the execution of jobs requiring a $slowdown$ value equal to 1. It can be seen that when the resources are not longer available the Backfilling and Flexible backfilling algorithms are not able to guarantee this QoS. The proposed class scheduler, by using the technique of resource stealing, makes available the resources needed also when the system contention is high ($RRI \geq 1.5$). It is worth to point out that the Flexible backfilling algorithm maintains the slowdown value within an acceptable level, offering in this test a performance comparable to the one obtained by the proposed scheduler.

6. Conclusion

In this paper, we propose a new multi-criteria scheduler to dynamically schedule a continuous stream of batch jobs on large-scale non-dedicated computing farm made of heterogeneous, single-processor or SMP machines, linked by a low-latency, high-bandwidth network. The proposed solution aims at scheduling arriving jobs respecting several functional and non-functional job requirements and optimizing the hardware and software resource usage. Several configuration parameters allow the scheduler customization with respect to the goals of an installation. The scheduler was evaluated by simulations using different job streams synthetically generated. To conduct the evaluation a technique to measure the system contention throughout a simulation was adopted. The scheduler has been evaluated comparing it with Backfilling and Flexible backfilling schedulers. In the conducted tests, the proposed scheduler demonstrated to be able to carry out good scheduling choices As future work, we plan: (1) to enhance the current scheduler refining the adopted advanced resource reservation technique, and to manage jobs requiring co-allocation to be executed on more than one machine, (2) to introduce energy efficiency policies dispatching workloads to more energy-efficient machines, (3) to evaluate the scheduler when applied to computing platforms made of distributed computing farm, (4) to investigate the feasibility of different scheduling criteria to estimate the RRI index.

7. Acknowledgment

This work has been supported by the Projects CONTRAIL (EU- FP7-257438) and S-CUBE (EU-FP7-215483).

References

[1] G. Capannini, R. Baraglia, D. Puppin, L. Ricci, and M. Pasquali. A job scheduling framework for large computing farms. In *SC*, page 54, 2007.

[2] P.-F. Dutot, L. Eyraud, G. Mounié, and D. Trystram. Bi-criteria algorithm for scheduling jobs on cluster platforms. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 125–132, New York, NY, USA, 2004. ACM.

[3] H. El-Rewini, T. G. Lewis, and H. H. ALI. *Task Scheduling in Parallel and Distributed Systems*. PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[4] Y. Etsion and D. Tsafrir. A short survey of commercial cluster batch schedulers. Technical Report 2005-13, School of Computer Science and Engineering, The Hebrew University of Jerusalem, May 2005.

[5] D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling—a status report. In *Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer, 2005.

[6] D. Klusek, H. Rudov, R. Baraglia, M. Pasquali, and G. Capannini. *Comparison of multi-criteria scheduling techniques, In: Grid Computing Achievements and Prospects*. Springer, 2008.

[7] K. Kurowski, J. Nabrzyski, A. Oleksiak, and J. Weglarz. A multicriteria approach to two-level hierarchy scheduling in grids. *J. of Scheduling*, 11:371–379, October 2008.

[8] Y. Kurowski, J. Nabrzyski, A. Oleksiak, and J. Weglarz. Scheduling jobs on the grid–multicriteria approach. *Computational Methods in Science and Technology*, 12(2):123–138, 2006.

[9] A. Mu'alem and D. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *Parallel and Distributed Systems, IEEE Transactions on*, 12(6):529–543, 2001.

[10] U. Schwiegelshohn and R. Yahyapour. Analysis of first-come-first-serve parallel job scheduling. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 629–638. Society for Industrial and Applied Mathematics, 1998.

[11] M. Siddiqui, A. Villazón, and T. Fahringer. Grid capacity planning with negotiation-based advance reservation for optimized qos. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 103. ACM, 2006.

Load Balancing Approach Based on Limitations and Bottlenecks of Multi-core Architectures on a Beowulf Cluster Compute-Node

Damian Valles, David H. Williams, and Patricia A. Nava
Electrical and Computer Engineering, University of Texas at El Paso, El Paso, Texas, USA

Abstract - *This paper studies the improvement of performance and execution time of a single compute-node in a Beowulf cluster. We want to implement a load balancing approach through the Linux scheduler which improves the performance and execution time of High Performance Linpack (HPL) benchmark. We compare the performance and execution time when spawning processes for two processing cores in a local processor up to all eight cores in two processors. The results showed that this approach helped to improve performance throughput since the load balancing approach created a higher L2-cache awareness, with increased hit rate, while reducing the number of times processes accessed the Front-side Bus (FSB) and Memory Controller Hub (MCH) during execution. Performance and execution time peaked with block sizes of 64 and 128 for different HPL matrix size and problem sizes; however, the performance throughput decreased for other sizes due to hardware contentions in the FSBes and MCH.*

Keywords: cluster, performance, quad-core, benchmark[1]

1 Introduction

Beowulf clusters are scalable performance clusters constructed with off-the-shelf hardware components, communicating on a private system network, with an open source software infrastructure [1]. A Beowulf cluster consists of a front-end machine that communicates with the outside world, and manages and distributes jobs to identical compute-nodes. The compute-nodes are connected to the front-end via a private network that normally uses a commonly available communication protocol such as Ethernet or InfiniBand for communication. Together, the compute-nodes and associated private network form a homogenous environment that make-up the structure of the cluster. The compute nodes are servers that are always listening for incoming requests and provide the necessary processing computational power to the assigned tasks coming from the front-end machine.

[1] This work was supported by National Science Foundation under Grants No. CNS-0709438 and CNS-1059430. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Virgo 2.0 is a Beowulf cluster system that consists of a front-end machine, twenty-one compute nodes and two memory nodes [2]. Throughout Virgo 2.0, the processing architecture is constant in all compute-nodes that consist of two Intel Quad-core Xeon processors [4]. Both processors can communicate with each other and to the rest of the system by their perspective Front-side Bus (FSB) into the Memory Controller Hub (MCH). The focus is to analyze the architecture of the two processors in this type of cluster environment in order to prevent performance bottlenecks that might be present during execution.

Benchmarking was utilized to measure the performance and execution time of the system in order to come understand the architecture [3]. High Performance Linpack (HPL) 2.0 was utilized in order to perform the benchmark on a single compute-node and spawn the desired number of processes to the cores. For the subprogram support of HPL, the Automatically Tuned Linear Algebra Software (ATLAS) 3.8.2 was installed which carries the Basic Linear Algebra Subprograms (BLAS) for the HPL algorithm to execute properly. And MPICH 2.0 was installed in order to carry the Message Passing Interface (MPI) standard libraries for proper communication between the processors and cores.

Assuming that the Linux Scheduler is applied to all compute-nodes in the cluster system, it is developed for its specific instruction set for the hardware, and it is the only mechanism that schedules all of the tasks that are executed for all users and applications. We hypothesize that the scheduler can be modified to further take advantage of the hardware configuration, in being more aware of the pre-existing limitations to the load balancing caused by specific hardware limitations, and by redefined the decision-making when performing load balancing of tasks. An average of double digit percentage improvement in performance and execution time for large computational jobs executed on Virgo 2.0 can be obtained.

2 Virgo 2.0 Overview

Virgo 2.0 is a Beowulf cluster that consists of a front-end machine, twenty-one dedicated compute-nodes and two memory-nodes. All compute-nodes consist of homogenous architecture parts which include two Intel Quad-core Xeon processors without hyper-threading capabilities. In effect, the

cluster can have up to one hundred and eighty-four processing cores working on a single application not including the front-end machine. Therefore, it gives the opportunity to explore the possibilities of improving the load balancing of tasks on Virgo 2.0 through reprogramming of the Operating System (OS) kernel to improve performance and execution time.

Virgo 2.0 is a cluster system in which a user has access to all compute-nodes once the user accesses the front-end machine. The user is able to launch individual and/or concurrent applications on each compute-node. If distributed applications are needed, the user must employ a communication protocol to spawn the processing to all needed compute-nodes. Such protocols include sockets; Remote Procedure Calls (RPCs) and the Message Passing Interface (MPI). All scheduling and job creation is interactive. This cluster system does not contain an internal batch queue program that launches and distributes jobs to its compute-nodes. However, the configuration of the cluster system provides an opportunity to improve performance of a single compute-node and replicate to all others.

Two Harpertown processors [4] is the architecture implemented on each compute-node with a total of eight processing cores. Each of the processing cores has their own Level 1 (L1) cache memory and each pair shares a 6MB Level 2 (L2) cache memory. The pair of processing cores also shares a FSB which provides the communication path for data and address buses.

Each of the Intel Quad-core Xeons is labeled with a physical identification number as shown in Figure 1. The physical identification given to the processor helps the system to identify the processing cores with even numbers assigned to processor physical id zero and odd numbers assigned to processor physical id one.

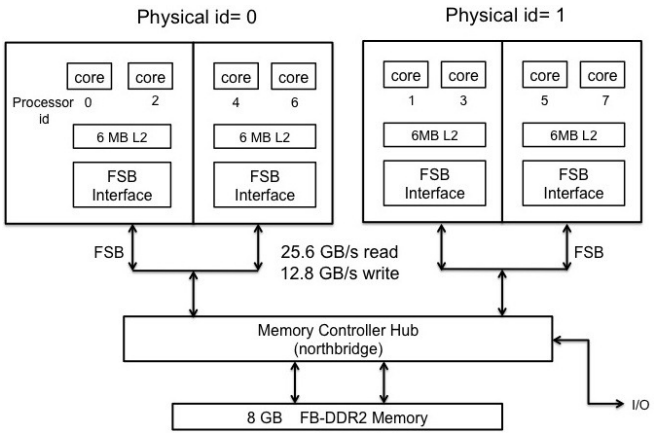


Figure 1. Harpertown Block Diagram Configuration in a Computer-Node

Each processor combines the signals from the two FSBs into a single dedicated FSB bidirectional interface to the Memory Controller Hub (MCH). Each of the FSBs has a 38-bit address bus, a 64-bit data bus, and associated control signals. The setup time for actually requesting the address/data buses is four FBS-clock cycles and the request of the busses can last up to a maximum of twenty FBS-clock cycles for that one single request [4], if the four clock cycles are not enough for the task to complete before another request must be issued.

As we try to improve the performance and management of tasks within the compute-node, we must consider the functionality and design of the FSBs. The merger of data from the two FSBs within each Harpertown and the merger of data of the FSB interfaces from the two processors within the MCH must be considered as tasks normally access the addresses and data from memory many times. What we can expect is that the larger the tasks become and the more memory accesses are needed, bottlenecks will start to occur as more and more data is moved through the FSBs and the MCH to each pair of processing cores within the Harpertown processor.

The Memory Controller Hub (MCH) is the main chipset that bridges both of the Harpertown processors together and is the pathway to the I/O Controller Hub. The MCH provides two FSB processor interfaces, four fully buffered DIMM (Dual In-line Memory Module) memory channels, nine x4 PCIe (Peripheral Component Interconnect Express) bus interfaces configurable with x8 or x16 ports, an Enterprise South Bridge Interface (ESI), six SM Bus interfaces for system management, and DIMM Serial Presence Detect (SPD) [4]. Figure 2 show a block diagram of the MCH with all of its interfaces.

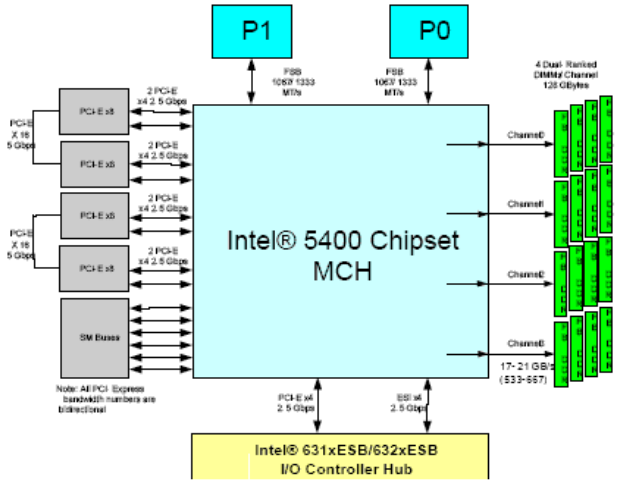


Figure 2. Intel's 5400 Chipset Memory Controller Hub Block Diagram

The MCH can play a big role when developing a load balancing mechanism of processes. One of the main decisions of the scheduler is to know when it is appropriate to migrate tasks from one core to another. It may not be a problem to

move the task to a neighboring core within the same die since all of the resources of the task are located in the same area. However, the decision can be expensive if the migration occurs from one core to another core in a different processor. This is because the task and all of its resources are too big to migrate completely within a single request and therefore forces a context switch. The migration can occur but the new destination might not have all the necessary space to accommodate all the migrating resources. That is why there must be an understanding in how the MCH can treat this scenario of task migration between two processing cores on different die.

Completely Fair Scheduler (CFS) is a scheduling algorithm that will run each process for some amount of time in round-robin order, then context switch, selecting next the process for execution that has run the least. Rather than assign each process a timeslice, CFS calculates how long a process should run as a function of the total number of runnable processes. And instead of using the nice value to calculate a time slice, CFS uses the nice value to weight the proportion of processor time a process is to receive: Higher valued (lower priority) processes receive a fractional weight relative to the default nice value, whereas lower valued (high priority) processes receive a larger weight [6].

The O(1) Scheduling algorithm is the current scheduler on the Linux 2.6.18 kernel. The big-O notation is used to denote the growth rate of an algorithm's execution time based on the amount of input [7]. Previous schedulers contained O(n) algorithms which indicated that the algorithm increases linearly or more as the input grows in size n. For example - the running time of $O(n^2)$ grows quadratically. Therefore, an O(1) algorithm is one that guarantees to operate at a constant time independently of the load size coming in to the input. The Linux 2.6.18 O(1) Scheduler includes per-core structures called run-queues, which contain the set of runnable threads assigned to each core in the MCA environment [8].

This becomes ideal for the Virgo 2.0 system because as the number of tasks increases for execution, the scheduler will be able to handle more tasks to be scheduled without additional system overhead. The idea of implementing a CFS Scheduler for the kernel would be unnecessary since none of the compute-nodes have any interaction processes to execute at any time. Therefore in combining the O(1) Scheduler algorithm and the SCHED_FIFO policy of each of the tasks executing on each compute-node, will help to improve system performance. SCHED_FIFO tasks do not have time slices and they may run indefinitely, which is desirable by avoiding preemption and increase processing core utilization. A SCHED_FIFO task can only be interrupted or preempted by other FIFO tasks that have higher priority or by system interrupts [7].

3 Procedure

The load balancing modification of the Scheduler deals with how the processes will be balanced and affinity assignments is made to match tasks with the processing cores. Assigning affinity of processing cores to each of the tasks immediately starts. The manner in which each task is assigned to a processing core is implemented using the load_aff() function.

The load_aff() function is a modification that load balances the tasks that are ready to execute and assigns the task's affinity to a processing core. The load_aff() function is as follows:

• It first determines if the previous modification was able to group single or multiple tasks for execution, if not, the function returns control as shown in the code below.

```
if(arraypid[0][0]==NULL)
  return;
```

• Next, the load_aff() function performs an availability test of the processing cores through the idle cpu(int cpu) function. The processing cores that are idle and ready for execution are saved in a vector array in order to use the list during the load balancing part of the function.

```
for(go through processing cores)
  ready[] = idle_cpu(order[]);
```

• The ready[] array keeps the information of the available processing cores that are available for execution.

• The order[] array is a defined array which holds the sequence of processing core identification numbers.

• The order[] array is ordered by pairs of processing cores that share the L2 cache memory.

After initial checks, the load_aff() function begins load balancing the tasks that were grouped in the previous modification and begins assigning affinity to processing cores. The simplest form of load balancing is when the number of tasks equals or is greater than the number of processing cores available for execution. In this case the load_aff() function will:

• Check that the order[] array and the ready[] array have the same number of elements and value of each element in order.

• For each element, the function begins to assign affinity to each task by their PID and the order of the processing cores.

• The pseudo-code below demonstrates in how tasks are assigned in this scenario.

```
if(check readyarray == orderarray)
 for(arraypid[][])
  if(tskcore=
    sched setaffinity(p->arraypid[][],
    cpumask of cpu(ready[])) == 0)
    if(ready[indx++]!= NULL)
     indx++;
    else return;
  else return -1;
```

• tskcore is an int-type variable which checks the return value of the sched_setaffinity() function.

• The sched_setaffinity() function is a defined function in the Scheduler which is used to set any task to a CPU. Each time the sched_setaffinity() function is called, it must also have the two parameters: the PID of the task and the mask-type value of the CPU.

• Each PID of tasks in the group is obtained from the arraypid[][] array used in the grouping of tasks modification.

• For the second parameter, each processing core ID that is in the ready[] array must be changed to cpumask_t type.

• The function cpumask of cpu(int cpu) is used to do this conversion correctly. The cpumask of cpu(int cpu) is a defined function in the Scheduler. The int cpu value is obtained from the ready[] array.

• If the sched_setaffinity() function returns a zero, the task affinity will be set to the indicated processing core.

• Also, it needs to keep track of the indexing of the ready[] array. It will check if it is at the end of the array.

This keeps repeating, setting affinity to each task, until all are assigned. However, many other scenarios of processing core availability can be presented when trying to execute a program with eight tasks. The load balancing would need to decide where the tasks in the group need to execute and assign affinity to all. Therefore, the load balancing must take into account the number of processes each instance of the HPL benchmark is asked to execute and decide the best affinity assignment accordantly when the number of tasks is greater than the available processing cores.

Once the number of concurrent threads continues to increase, the IPC performance and the L2 hit rate begin to decrease. Therefore, the load_aff() function must consider performance decreases that can occur by assigning affinity to all the processing cores. The way the load balancing is modified is by setting affinity in pairs of two processing cores that share L2 cache memory. The main reason to pair the processing cores in this manner is to take advantage of the L2 cache share capabilities. Secondly if any process communication is needed, the tasks will have minimal latency. Finally, as the processes share L2 cache memory to execute the same instance of the benchmark, it reduces the number of L2 miss rates which in effect reduces the number of main memory accesses and creates less contention in the FSB and MCH.

The load balancing decision making is shown in Figure 3. The decision making begins with finding the number of available processing cores and servicing the number of tasks that are ready to execute. If the number of processing cores is greater than or equal to the number of tasks, the function will set affinity for all tasks and then continue scheduling. However if the number of processing cores are less than the number of tasks that are ready to execute, then the function must check the number of available processing cores and begin deciding the course of action that can be taken as follows:

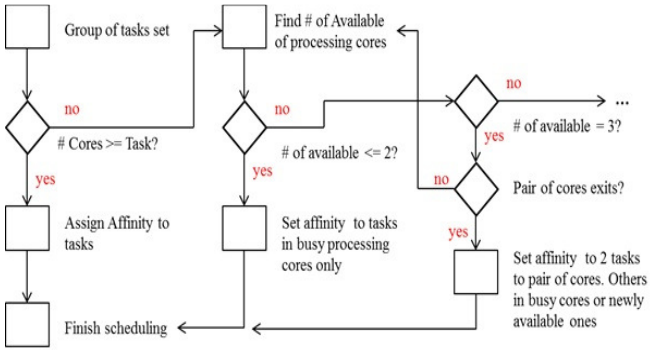


Figure 3. Low Balancing Flowchart

• Once the number of available of processing cores is less than the number of tasks, the function checks if the number of available processing cores is less than or equal to two.

• If the number of processing cores available is less than or equal to two, the decision is to not set affinity to the available processing cores in order to:

- Avoid using all of the processing cores which causes a decrease of IPC performance.
- Lowers the overall L2 hit rate for all processing cores.
- Avoid increase of traffic through the FSBuses and MCH.

• Instead, the function will set affinity to processing cores that are currently busy in order to schedule them next.

• This keeps the available processing core free and next group of tasks out of the way of the currently execution of tasks.

When checking that the number of available of processing core exceeds two, then the number can be either an odd or even number of available processing cores. The decision of the load balancing will need to consider the availability of processing cores in pairs and if no pair exits, then continue to wait for processing cores to become available.

The configurations of the HPL benchmark used for current test are similar to previous work [3] but each run contained a

heavier load. In where, N – The number of problems sizes that will be used for execution of the benchmark. Ns – This specifies the exact value(s) of the problem size. NBs – It specifies the values of each block size to be executed for each problem size defined in Ns. Ps – Specifies the number of processes row value. Qs – Specifies the number of processes column value.

Eight different problem sizes Ns = {6000, 7000, 8000, 9000, 11000, 12500, 15000, 17500} were used for each instance of the benchmark. From [3], it could be seen that the benchmark provided better data with problem sizes higher than 5000. The matrix dimensions were varied for different Ps and Qs values and also depended on the scenario that needed to be tested (i.e. all eight cores, seven cores, down to just two cores). Block sizes NBs = {64, 128, 256, 512, 1024, 2048} were used for each of the problem sizes.

Different instances of the HPL benchmark are necessary since it introduces an environment that tests the modifications made to the Scheduler. Each instance of the benchmark can spawn other processes that will test the grouping of the processes with the parent process. After each instance of the benchmark finished execution, process termination and the dead state of the parent process was tested to determine if revising the state of the processing core was possible in order to assign affinity to the next group of tasks that are ready to execute.

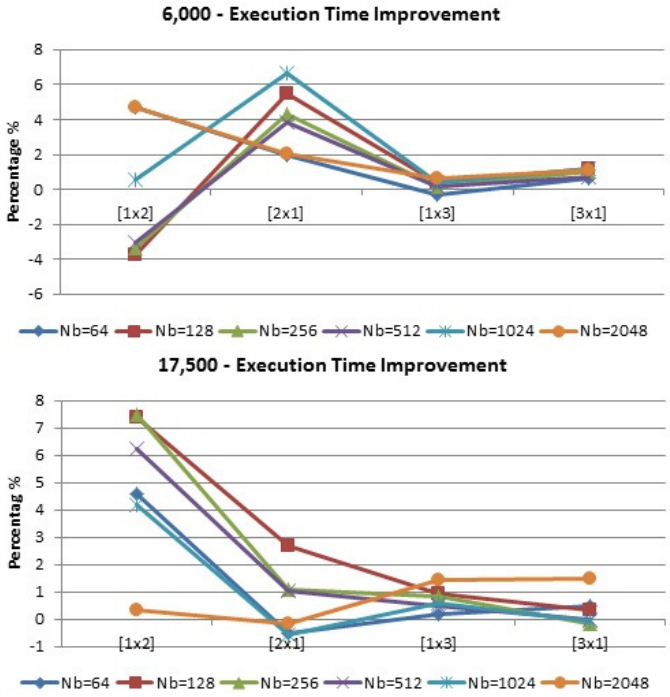


Figure 4. Execution Time Percentage Improvement in a Four Process Execution

4 Results

The low concurrency case was expanded to include up to and including four processes. This qualifies as a low concurrency case because it provides enough concurrent processes to create small contentions in the FSB and MCH and test the Scheduling modifications. This number of concurrent processes also illustrates higher number of HPL benchmark instances that were launched to test low concurrency scenarios. Figure 4 summarize the percentage improvements of execution time for this scenario for problem sizes 6000 (smallest problem size) and 175000 (largest problem size). The improvements are listed by different matrix dimensions and different block sizes during execution.

Figure 5 shows the improvement percentage of performance throughput in GFLOPS of the same problem sizes shown in Figure 4. The percentage improvements from Figures 4 and 5 indicate it is hard to improve the performance and execution time when the processing core executes small loads. This is due to several factors:

1. When the computational load does not employ all of the available processing cores, the performance of the overall compute-node does not reach its maximum performance. This is true even with customized load balancing and affinity task allocation.

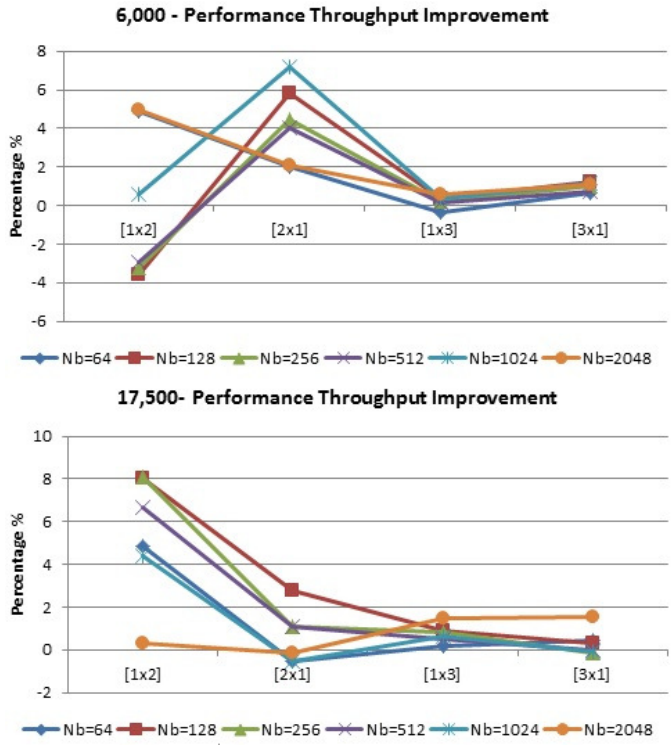


Figure 5. Performance Throughput Percentage Improvement in a Four Process Execution

2. The small improvements that were attained are due to the load balancing modification, which provided task affinity, which allowed the processing cores to share L2-more efficiently. This is achieved by the decision making of having tasks allocated with processing cores

Spreading out the tasks between the two processors alleviates traffic in both FSBs; and each task has an increase amount of L2-cache since not all cores are in use. The following case shown in Figures 6 and 7 is of high concurrency with eight concurrent processes occupying all the processing cores. The configuration consists of having two or more instances of the HPL benchmark running in which the total of processes equal to the number of all the processing cores. More than one instance of the benchmark is needed to make the Scheduler group and load balance all processes.

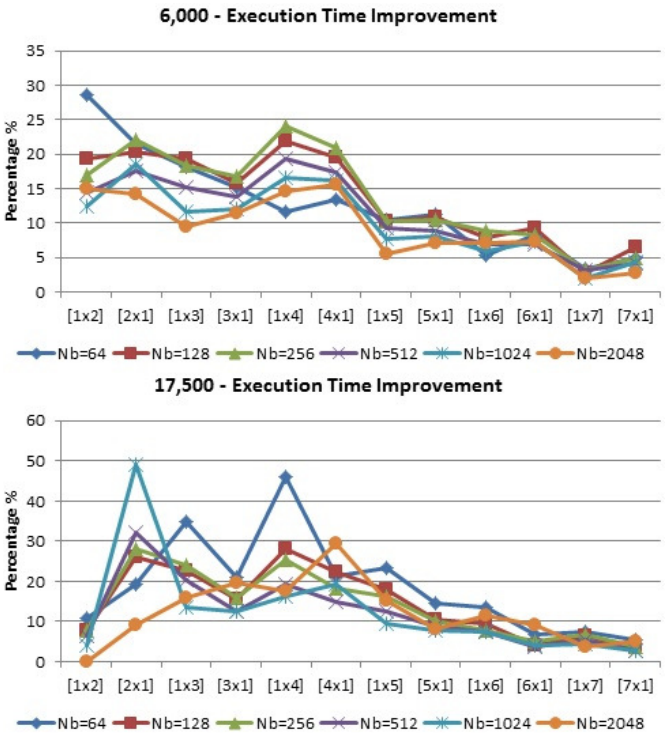


Figure 6. Execution Time Percentage Improvement in an Eight Process Execution

In order to summarize the percentage improvements of the eight concurrency execution scenario, Figure 6 shows the improvement in execution time of problem sizes 6000 (smallest problem size) and 175000 (largest problem size). The improvements are showed by different Nb block sizes, each using different PxQ matrix dimensions during execution. Figure 7 shows the improvement percentage of performance throughput, in GFLOPs, for the same problem sizes shown in Figure 6. The improvements are showed by different matrix dimensions each through different block sizes during execution. The percentage improvements from Figures 6 and 7 indicate that for low PxQ matrix dimensions, there is greater improvement in performance and execution time. However, it is much harder to improve the performance and execution time when executing with all of the processing cores. This is due to several factors:

1. The other instances of the benchmark concurrently executing increases the traffic in the FSB and MCH.

2. As the problem size increases, the limitations of the hardware (bus sizes, cache sizes) begin to decrease processing core utilization.

3. As the bottlenecks and limitations become more apparent during execution, the modifications made to the Scheduler converge to very low to no improvement regardless of the affinity set to all tasks.

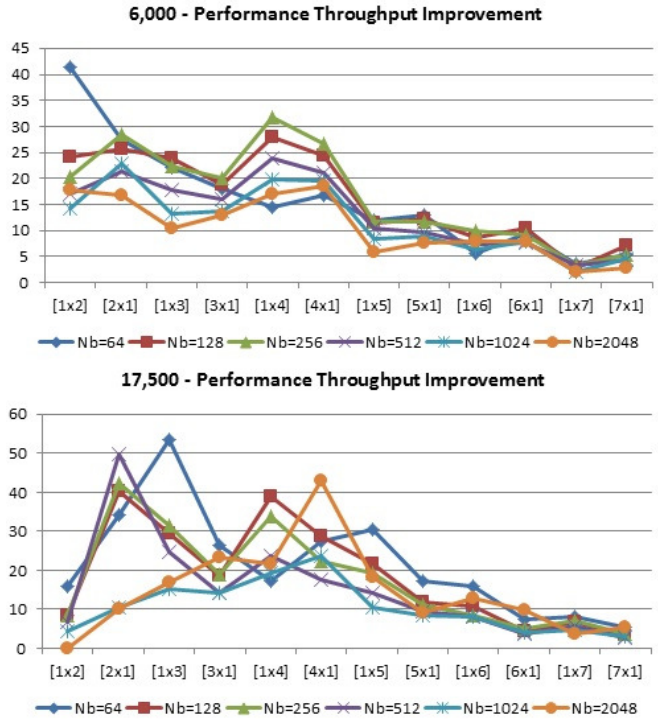


Figure 7. Performance Throughput Percentage Improvement in an Eight Process Execution

5 Conclusions

Overall, the results for low concurrency show that the improvements in both GFLOPS and execution times were less than 10% for low numbers of processing cores. Regardless of the block sizes, the small improvements are due to the affinity assignment which help increase the L2 cache hit rates. However, the affinity assignments can sometimes create contention to the FSB when load balancing other instances of the benchmark which can be seen by the negative percentage numbers at different problem sizes in Figures 4 and 5. With larger numbers of processes (dimensions) performance cannot be improved since FSB and MCH contention increases; in fact, the GFLOPS decreases and execution times increases.

Overall the results for medium to high concurrency show that the larger improvements were obtained by these common factors:

• Small block sizes of 64-128 show the highest improvements than other block sizes due to the width of the data line that exists in the hardware.

• The larger the block sizes, the harder it is to improve GFLOPS and execution time due to the time to transmit information; although improvements were still achieved with smaller percentages from the smaller block sizes.

• Small numbers of processes (dimensions) are easier to allocate affinity to processing core pairs. In this case the load balance modification helps increase the processing core utilization which yields a higher improvement of GFLOPS.

• Larger number of processes (dimensions) are difficult to speed up as contention on the FSB and MCH increases; therefore, the GFLOPS decrease and execution time increases.

• All of the benchmark runs and instances were executed in row-major format. This means that the elements of the matrix are placed in a contiguous form in memory. Therefore, single row dimensions of the benchmark show higher peaks in GFLOPS and execution time.

A few cases of large peaks indicate high percentage improvements in GFLOPS and execution time, such as in the bottom figure in Figure 7 where matrix size dimensions are [1x3] at block size 64 or similar dimensions. This is due to a non-modified run under-performed in this instance of the benchmark which causes the modified run to show higher increases in percentage in GFLOPS and execution times. Also, the matrix dimensions are favorable for the row-major execution of the matrix during each run. Furthermore, the small block size that is able to use the full bandwidth of data lines and the L2 cache awareness which helped improved hit rates for the instance of the benchmark.

Other cases of large peaks that indicate high percentage improvements in GFLOPS and execution time, such as in the bottom figure in Figure 7 where matrix size dimensions [4x1] at block size 2048 or similar dimensions are attributed to the non-modified run under-performing which causes the modified run to show higher increases in percentage in GFLOPS and execution times. Also, the load balancing modification helped increase the percentages due to the affinity assignments of the instance of the benchmark.

6 References

[1] Beowulf Cluster Website, "Beowulf Cluster Overview", 2008. [Online]. *http://www.beowulf.org* [Accessed: September 2008].

[2] D. Valles, "Development of Load Balancing Algorithm Based on Analysis of Multi-core Architecture on a Beowulf Cluster," Ph.D. Elec. And Computer, Dissertation, University of Texas at El Paso, El Paso, TX, Dec. 2011.

[3] D. Valles, D. Williams, P. Nava, "Performance and Timing Measurements in a Multi-core Beowulf Cluster Compute-Node," in *Proc. PDPTA'09*, Las Vegas, NV, 2009.

[4] Intel Corporation, "Intel 5400 Chipset Memory Controller Hub (MCH)," *Datasheet*, Document Number: 318610, November 2007.

[5] Official Rocks Clusters website, "Rocks Cluster Distribution: Users Guide," 2008. [Online]. *http://www.rocksclusters.org/rocksdocumentation/4.3/* [Accessed: June 2008]

[6] Robert Love, *Linux Kernel Development*, 3rd ed., Crawfordsville, IN: Addison-Wesley, 2010.

[7] Josh Aas, "Understanding the Linux 2.6.8.1 CPU Scheduler," Silicon Graphics Inc., February 2005.

[8] S. Ziemba, G. Upadhyaya, V. Pai, "Analyzing the Effectiveness of Multicore Scheduling Using Performance Counters," in *Proc. WIOSCA, in conjunction with ISCA-35*, June 2008.

A Simulation Study of Cooperative Load Balancing in Central-Server Node Distributed Systems

Satish Penmatsa and Jiromu Amioku

Department of Mathematics and Computer Science
University of Maryland Eastern Shore
Princess Anne, MD 21853, USA

Abstract - *Load balancing is very important for achieving high performance in distributed computer systems which often consist of heterogeneous computing and communications resources. In this paper, we study a cooperative load balancing scheme for central-server node distributed systems (CCOOP-IO) and evaluate its performance using simulations. The objective of CCOOP-IO is to minimize the mean response time of jobs in a heterogeneous distributed computing system and also to provide fairness to all the jobs in the system. We consider a heterogeneous computing system model connected by a single-channel communications network. A central-server model is used to model the computers in the system. The performance of CCOOP-IO is evaluated using simulations with various system loads and configurations.*

Key words: Load balancing, Resource Allocation, Distributed Systems, Fairness.

1. Introduction

The computing resources (computers or nodes) in distributed computing systems are often heterogeneous. Jobs may arrive with different job arrival rates to these nodes. Also, the communications networks that connect the computing resources may have different bandwidths. The above factors may degrade the performance of distributed systems if load is not properly balanced among the computers. Hence, load balancing is very important for achieving high performance in distributed computer systems.

The problem of load balancing in distributed systems has been studied extensively. For example, in [4, 11, 15, 16, 10], static load balancing schemes for single-class and multi-class job distributed systems were proposed and analyzed by considering various network topologies. Various models for dynamic load balancing were studied in [1, 6, 14]. A macroeconomic model for resource allocation in distributed systems was studied in [12]. Most of the past work on load balancing in distributed systems considered the minimization of the overall system expected (mean or average) response time (job execution time) as their main objective. However, some jobs might experience much longer response time than the others in such allocations. Providing *fairness* to all the jobs in the system is to find an allocation of jobs to computers that yields an approximately equal expected response time for all the jobs of approximately the same size. Fairness is a major issue in many modern computing systems.

Load balancing in distributed systems based on game theory with the objective of providing fairness has been studied ([2, 5, 7, 17] and references there-in). However, in most of the above studies, the computer model considered has only a processor. Game-theoretic scheduling in cognitive radio systems has been studied in [13] and references there-in. Here, we consider a central-server computer model which is very common in modern computer systems. A central-server computer model consists of a CPU (processor) and one or more input/output (I/O) devices. A central-server node model for static job allocation in E-commerce systems and utility-computing systems has been studied in [8, 9] and for dynamic job allocation in [6].

In this paper, we study a cooperative load balancing scheme for central-server node distributed systems (CCOOP-IO). CCOOP-IO is derived from the cooperative scheme studied in [7]. The objective of CCOOP-IO is to minimize the mean response time of jobs in a heterogeneous distributed computing system and also to provide fairness to all the jobs in the system. We consider a heterogeneous computing

system model connected by a single-channel communications network. Jobs arrive at each computer according to a time-invariant exponential process. We achieve load balancing by transferring some jobs from the heavily loaded nodes to the nodes that are idle or lightly loaded.

The performance of CCOOP-IO is evaluated using simulations with various system loads and configurations. *Expected response time* (execution time) and *fairness index* are used as the performance metrics. For comparison, we also implemented two representative load balancing schemes. These schemes are: OPTIM-IO (which minimizes the expected response time of all the jobs in a system) and PROP-IO (which allocates the jobs to the computers in proportion to their processing speeds in the system).

2. Cooperative Load Balancing

A distributed computing system model having *n* nodes connected by a single channel communications network is considered. The nodes in the system are typically heterogeneous having different processing speeds. Each node is modeled as a central-server model as shown in Figure 1 similar to [6]. The terminology and notations used similar to [3, 6, 7] are as follows:

- t_{IO}: The service time of an input/output (I/O) device.
- μ_i: The service rate of node *i*.
- ϕ_i: The external job arrival rate at node *i*.
- $\boldsymbol{\Phi}$: The total external job arrival rate of the system. So, $\boldsymbol{\Phi} = \sum_{i=1}^{n} \phi_i$.
- βi : The job processing rate (or load) allocated by the load balancing algorithm for node *i*.
- x_{ij}: The job flow rate from node *i* to node *j* (i.e. the number of jobs sent from *i* to *j* per unit time).
- *t*: Mean communication time for sending or receiving a job from one node to another.
- P_0: Probability that a job after departing from the processor finishes.
- P_1: Probability that a job after departing from the processor requests I/O service.
- P_1/P_0: Average number of I/O requests per job.

Each node is assumed to have a single computing resource (processor) with a round-robin service discipline and jobs arrive in a single queue. The nodes and the communications network have an exponential service-time distribution [3] and the external jobs arriving at each node and jobs being transferred by the communications network follow a Poisson distribution [3].

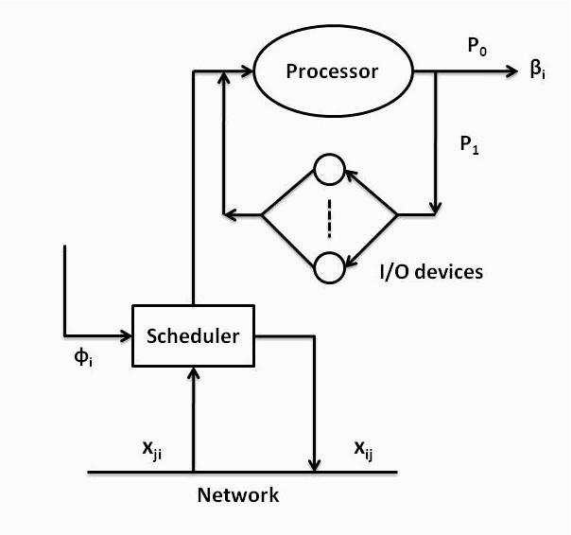


Figure 1. Node Model

A job arriving at node *i* may either be processed at node *i* or transferred to node *j* through the communications network for remote processing. The mean communication delay from node *i* to node *j* is independent of the source destination pair (*i*, *j*) but depends on the total traffic through the network denoted by λ where $\lambda = \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij}$. Based on the above assumptions and assumptions similar to [7], the mean node delay (mean response time or total execution time) of a job at node *i* is given by:

$$d_i(\beta_i) = \frac{1}{\mu_i - \beta_i} + \frac{P_1}{P_0} t_{IO}, \quad i = 1, \ldots, n.$$

The mean communication delay for a job is given by:

$$g(\lambda) = \frac{t}{1 - t\lambda}, \ \lambda < \frac{1}{t}.$$

The inverse of the node delay is given by:

$$d^{-1}(x) = \mu_i - \frac{1}{(x - \frac{P_1}{P_0} t_{IO})} \quad if \ x > \frac{1}{\mu_1} + \frac{P_1}{P_0} t_{IO}$$

$$d^{-1}(x) = 0 \quad if \ x \leq \frac{1}{\mu_1} + \frac{P_1}{P_0} t_{IO}$$

We also assume that the communication delay incurred as a result of sending a job directly from node *i* to node *j* is less than or equal to the sum of the delays from node *i* to node *k* and from node *k* to node

j. Based on this assumption, nodes are classified into Sinks (S), Idle Sources (R_d), Active Sources (R_a), and Neutrals (N) similar to [7].

The load balancing problem for providing fairness to all the jobs in the system is formulated as a cooperative game among the computers and the communications subsystem similar to [7]. Based on the Nash Bargaining Solution (NBS) which provides a *pareto* optimal and fair solution, we provide an algorithm (CCOOP-IO) for computing the NBS for our cooperative load balancing game.

In the following, we present the CCOOP-IO algorithm. The cooperative load balancing game among the computers and the communication subsystem, theorems, and properties which are the basis for the below algorithm are similar to the ones described in [7] by replacing *d*, *g*, and d^{-1} in [7] by *d*, *g*, and d^{-1} presented in the previous section.

CCOOP-IO Algorithm:

Input: Node job service rates: $\mu_1, \mu_2, \ldots, \mu_n$
Node job arrival rates: $\emptyset_1, \emptyset_2, \ldots, \emptyset_n$
Mean communication time: t
Service time of an I/O device: t_{IO}
Probabilities: P_0, P_1

Output: Load allocation to the nodes: $\beta_1, \beta_2, \ldots, \beta_n$

1. Initialize the loads of all the nodes to their job arrival rates and label all the nodes as *Neutrals*.
2. Sort the computers in increasing order of their node delays.
3. Categorize the nodes into Sinks (S), Idle Sources (R_d), Active Sources (R_a), and Neutrals (N) using a binary search (for finding an optimal point (say, α) that categorizes) similar to Step 3 of the CCOOP algorithm in [7].
4. Determine the loads on the computers as follows:
   - $\beta_i \leftarrow 0$, if node *i* is an Idle Source.
   - $\beta_i \leftarrow d^{-1}(\alpha + g(\lambda))$, if node *i* is an Active Source.
   - $\beta_i \leftarrow d^{-1}(\alpha)$, if node *i* is a Sink.
   - $\beta_i \leftarrow \emptyset_i$, if node *i* is a Neutral.

3. Experimental Results

In this section, we evaluate the performance of the CCOOP-IO scheme. The performance metrics that are used in the experiments are the *expected response time* and the *fairness index*. The fairness index [7] is used to quantify the fairness of load balancing schemes. We also implemented the Overall optimal load balancing scheme (OPTIM-IO) [4] and the Proportional load balancing scheme (PROP-IO) [1] for comparison purposes.

System utilization represents the amount of load on the system and is defined as the ratio of the total arrival rate to the aggregate service rate of the system. A heterogeneous distributed system consisting of 16 computers was simulated (as shown in Table 1) to study the effect of system utilization. The system has computers with four different service rates. For each experiment, the total job arrival rate in the system is determined by the system utilization and the aggregate service rate of the system. We had chosen fixed values for the system utilization and determined the total job arrival rates. The mean communication time is assumed to be 0.001 sec. We assumed that I/O operations were evenly spread throughout the execution of each job (similar to [6]) and that each disk I/O request took 0.06 milliseconds. The number of I/O requests for each job was chosen from a normal distribution with a mean of 12 and a standard deviation of 10 and was assumed to be greater than 0.

Table 1. System Configuration.

Relative service rate	1	2	5	10
Number of computers	6	5	3	2
Service rate (jobs/sec)	10	20	50	100

In Figure 2, we present the expected response time of the system for different values of system utilization ranging from 10% to 90%. The performance of CCOOP-IO is very similar to OPTIM-IO for system utilizations ranging from 10% to 40% and is around 50% better than PROP-IO for system utilizations ranging from 50% to 60%. CCOOP-IO approaches PROP-IO for high system utilizations.

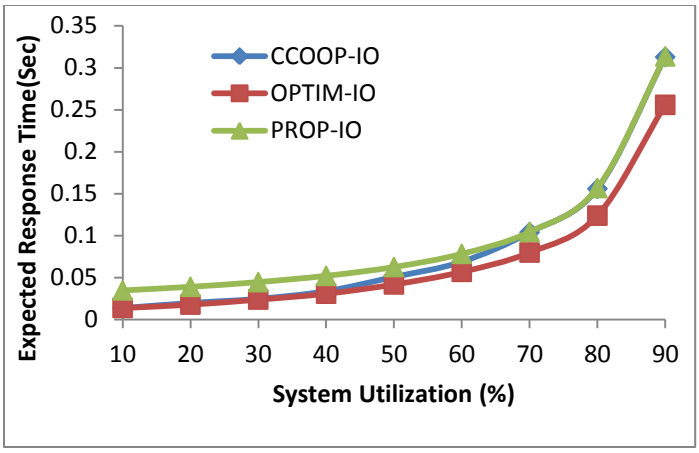


Figure 2. Expected Response Time v/s System Utilization

The effect of the system utilization on the fairness index of the various schemes is presented in Figure 3. It can be observed that the fairness index of CCOOP-IO is almost 1 for any system utilization and the fairness index of OPTIM-IO drops from 1 to around 0.85. PROP-IO has a constant fairness index which is around 0.72. This shows that CCOOP-IO provides fairness to all the jobs in the system independent of the computers to which they are allocated.

Figure 4 presents the expected response time at each computer for all the schemes at a system utilization of 70%. It can be observed that CCOOP-IO guarantees almost equal expected response times for all the computers. This means that all the jobs will have almost the same expected response time independent of the allocated computers. In the case of OPTIM-IO and PROP-IO, the expected response times are less balanced than CCOOP-IO.

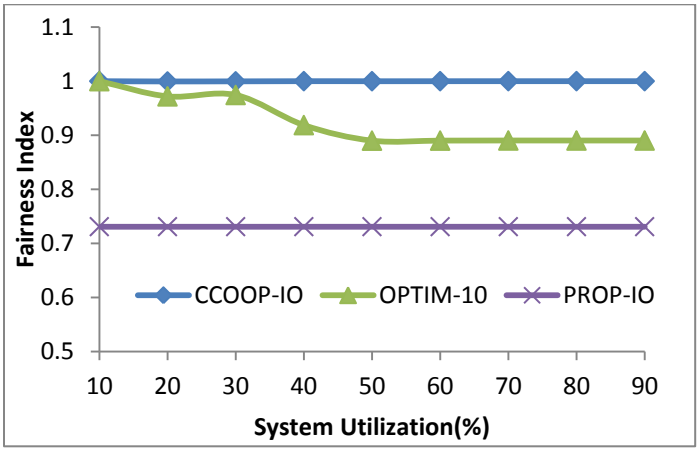


Figure 3. Fairness Index v/s System Utilization

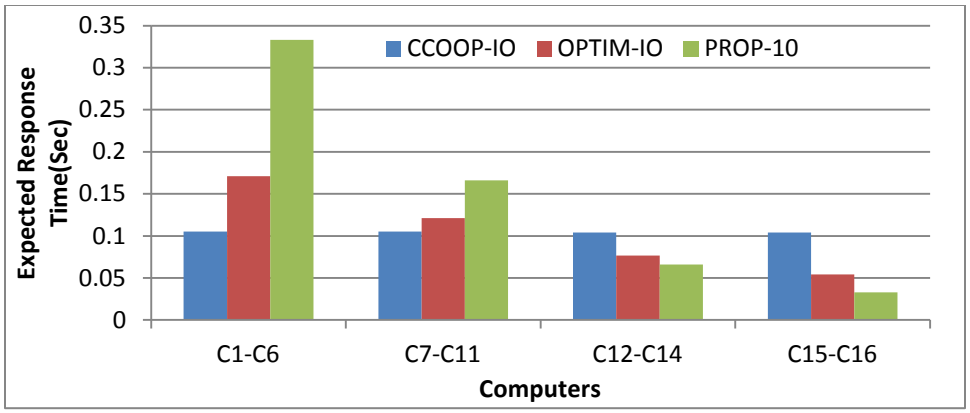


Figure 4. Expected Response Time at each Computer (System Utilization = 70%)

In the following, we study the effect of heterogeneity (speed skewness) [7] on the performance of CCOOP-IO. Speed skewness is defined as the ratio of maximum service rate to the minimum service rate of the computers in the system. A heterogeneous distributed system of 16 computers (2 fast and 14 slow) was simulated to study the effect of heterogeneity. Slow computers have a relative processing rate of 1 and the relative processing rate of the fast computers is varied from 1 (homogenous system) to 20 (highly heterogeneous system).

Figure 5 presents the effect of speed skewness on the performance of CCOOP-IO. For low skewness, the performance of CCOOP-IO is similar to PROP-IO. However, as the skewness increases, the performance of CCOOP-IO approaches to that of OPTIM-IO. Figure 6 presents the effect of speed skewness on the fairness index of CCOOP-IO. It can be observed that CCOOP-IO has a fairness index of almost 1 over all range of speed skewness. The fairness index of OPTIM-IO and PROP-IO falls from 1 at low skewness to 0.95 and 0.9 respectively at high skewness. This shows that CCOOP-IO provides fairness in highly heterogeneous systems for all the jobs in the system.

Figures 7 and 8 present the expected response time at each computer for all the schemes at medium system utilization for a skewness of 8 and 12. CCOOP-IO guarantees almost equal expected response times for all the computers. This means that CCOOP-IO provides a fair and load balanced allocation compared to OPTIM-IO and PROP-IO where the jobs are treated unfairly.

4. Conclusions

In this paper, a cooperative load balancing scheme (CCOOP-IO) for heterogeneous distributed systems was studied and evaluated. A distributed system with central-server nodes was considered. The performance of CCOOP-IO is evaluated by varying the system utilization and heterogeneity. Experimental results showed that CCOOP-IO is not only fair but also is comparable with that of the system optimal scheme in terms of the mean response time.

Acknowledgements

This research is supported in part by the Department of Mathematics and Computer Science at the University of Maryland Eastern Shore.

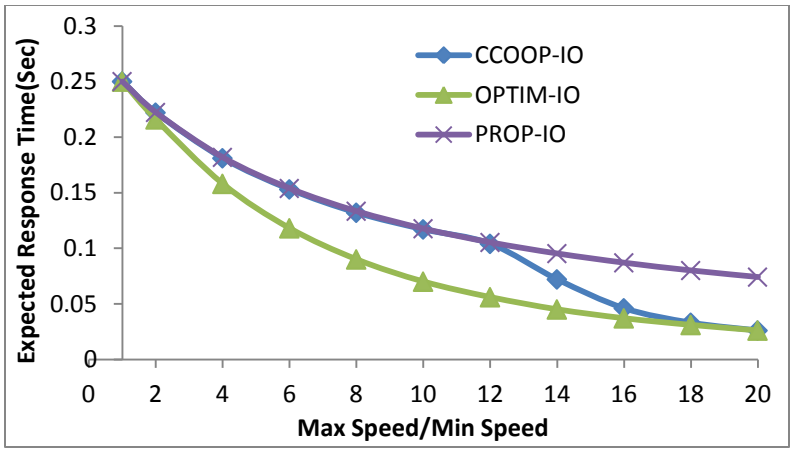


Figure 5. Expected Response Time v/s Heterogeneity

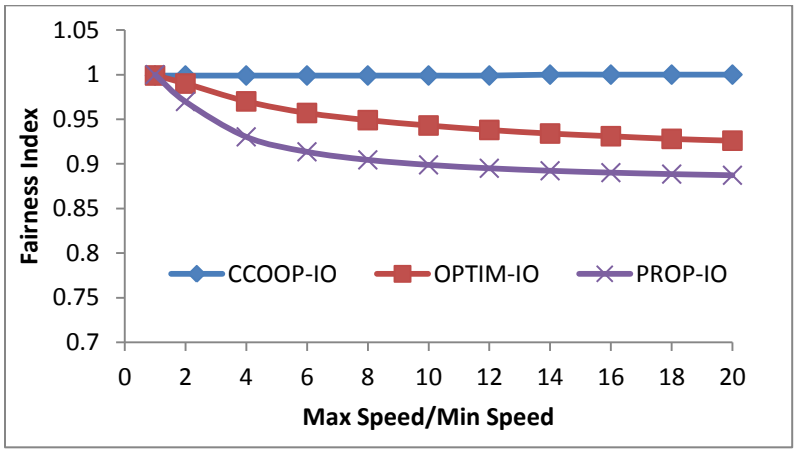


Figure 6. Fairness Index v/s Heterogeneity

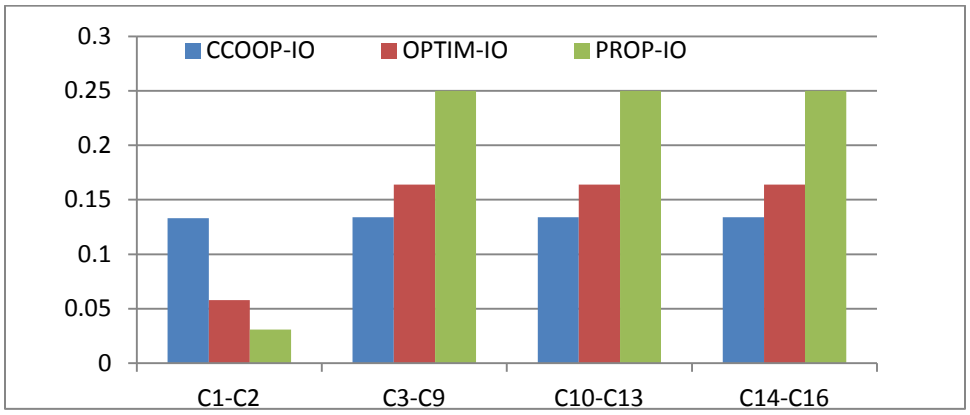


Figure 7. Expected Response Time at each Computer (Speed Skewness = 8)

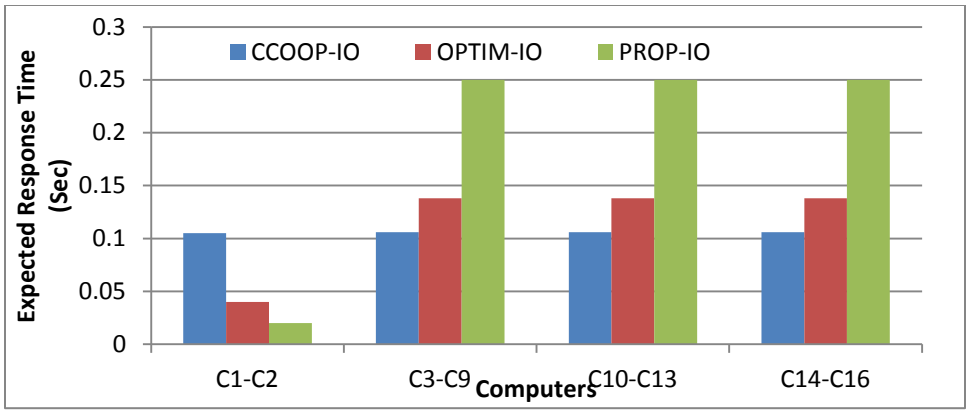


Figure 8. Expected Response Time at each Computer (Speed Skewness = 12)

References

[1] Y.C. Chow and W.H. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Trans. On computers*, C-28(5):354-361, May 1979.

[2] D. Grosu and A.T. Chronopoulos. Noncooperative load balancing in distributed systems. *Journal of parallel and distributed computing*, 65(9):1022-1034, Sep. 2005.

[3] R. Jain. *The Art of computer systems performance analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, 1991.

[4] H. Kameda, J. Li, C. Kim, and Y. Zhang. *Optimal Load balancing in distributed computer systems*. Springer Verlag, London, 1997.

[5] R. Subrata, A. Zomaya, and B.Landfeldt. Game theoretic approach for market-like computational grids. *IEEE Trans. On parallel and distributed systems*, 19(1):66-76, Jan. 2008.

[6] Y. Zhang, K. Hakozaki, H. Kameda, and K. Shimizu. A performance comparison of adaptive and static load balancing in heterogeneous distributed systems. In proc. *IEEE 28th Annual Simulation Symp.*, pages 332-340, Phoenix, AZ, 1995.

[7] S. Penmatsa and A.T. Chronopoulos. Game theoretic static load balancing for distributed systems. *Journal of Parallel and Distributed Computing*, 71 (2011), 537-555.

[8] S. Penmatsa and Gurdeep S. Hura. Job Allocation in E-Commerce Systems Involving Self-Interested Agents. *Journal of Global Information Technology*, 5(1), pp 1-10, 2010.

[9] S. Penmatsa and Gurdeep S. Hura. Cost Minimization in Utility-driven Distributed Computing Systems with Central-Server Model. *Proceedings of the 23rd International Conference on Computer Applications in Industry and Engineering (CAINE 2010)*, Las Vegas, NV, USA, Nov. 8-10, 2010.

[10] S. Penmatsa and V. P. Mantena. Comparison of Static Load Balancing Schemes for Utility-driven Distributed Computing. Proceedings of the 16th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2010), Las Vegas, NV, USA, July 12 15, 2010.

[11] A.N. Tantawi, D. Towsley, Optimal static load balancing in distributed computer systems, J. ACM 32 (2) (1985) 445–465.

[12] X. Bai, D.C. Marinescu, L. Boloni, H.J. Siegel, R.A. Daley, I.-J. Wang, A macroeconomic model for resource allocation in large-scale distributed systems, J. Parallel Distrib. Comput. 68 (2) (2008) 182–199.

[13] Qiang Ni and C. Zarakovitis, Nash Bargaining Game Theoretic Scheduling for Joint Channel and Power Allocation in Cognitive Radio Systems, IEEE J. on Selected Areas in Communications, 30(1), 2012, pp. 70-81.

[14] Y. Zhang, H. Kameda, S.L. Hung, Comparison of dynamic and static load balancing strategies in heterogeneous distributed systems, IEE Proc. Comput. Digit. Tech. 144 (2) (1997) 100–106.

[15] N.G. Shivaratri, P. Krueger, M. Singhal, Load distributing for locally distributed systems, Comput. 25 (12) (1992) 33–44.

[16] T.D. Braun, H.J. Siegel, A.A. Maciejewski, Y. Hong, Static resource allocation for heterogeneous computing environments with tasks having dependencies, priorities, deadlines, and multiple versions, J. Parallel Distrib. Comput. 68 (11) (2008) 1504–1516.

[17] K. Rzadca, D. Trystram, A. Wierzbicki, Fair game-theoretic resource management in dedicated grids, in: Proc. of the 7th IEEE Intl. Symp. on Cluster Computing and the Grid, Brazil, May 2007, pp. 343–350.

Performance Evaluation of a Dynamic Single Round Scheduling Algorithm for Divisible Load Applications

Leila Ismail, Liren Zhang, Khaled Shuaib, and Sameer Bataineh
Faculty of Information Technology, UAE University, 17551 Al-Maqam, Al-Ain, United Arab Emirates

Abstract— *Divisible load applications occur in many scientific and engineering applications and can easily be mapped to a distributed environment, such as computational grids or clouds, using a master-worker pattern. However, dividing an application and deploying it on a set of heterogeneous computing resources pose challenges to obtain an optimal performance due to the underlying system processing and networking capacities. We provide a dynamic scheduling algorithm for allocating divisible loads on a set of heterogeneous resources to decrease the overall application execution time. The algorithm uses a single-round strategy which is a known approach adopted by the majority of the developers as it is simple to design and implement. Our algorithm computes the chunk size that should be distributed to each worker. We analyze the performance of the algorithm in different scenarios of heterogeneous computing and networking capacities.*

Keywords: Distributed Systems, Divisible Load Application, Dynamic Scheduling, Performance

1. INTRODUCTION

The capacity of today's infrastructures, the ubiquity of network resources, and the low storage cost has led to the emergence of heterogeneous distributed memory systems such as Grid and Cloud computing. These platforms are promising technologies to run parallel applications in a low cost [1] [2] [3] [4] [5].

In this work, we propose a dynamic scheduling algorithm to obtain an optimal performance when distributing divisible load applications [6] to a set of heterogeneous resources in a computational platform, such as a Grid or a Cloud. Divisible load model represents a class of applications, where an application can be divided into a number of tasks that can be processed independently in parallel ([6], [7]) with no or negligeable inter-tasks communication. Many scientific and engineering applications fall into this category, such as search for a pattern, compression, join and graph coloring and generic search applications [8], multimedia and video processing [9], [10], convolution [11], and image processing [12] [13].

The tasks of a divisible load application are amenable to a master-worker model [14] that can be easily implemented and deployed on computing platforms, such as clusters, computational Grids, or computational Clouds. The master is a processor which divides an application load into tasks and assigns each task to a separate worker. In this work, we compute the chunk sizes of an application that should be distributed to a set of computing workers to obtain an optimal performance. The algorithm takes into consideration the communication and the computation capacities of the underlying platform. It also accounts for computing delays and communications latencies. In our scenarios, our algorithm shows a maximum of 1.29 of relative makespan to the *ideal* algorithm in the heterogeneous environments we have set, and with no specific selection policy. The *ideal* algorithm determines a chunk size in an ideal situation in which all computing workers are homogeneous, and that networking and computing overheads are negligeable.

Several works have been done to find an optimal scheduling algorithm to schedule a divisible load application [15] [16] [17] [18] [19]. The multi-installment algorithm, studied in [15], proposed a model of scheduling tasks in a single round, but communication time and computation time are assumed to be proportional. A multiround algorithm [16] was built based on [15] by adding communication and computation latencies to the model. The algorithm aims at optimizing the number of rounds to obtain an optimal makespan of the application. Consequently some computing workers will be waiting, while others are working within a single round. [20] studies the complexity of multi-round divisible load schedule, and concluded that the selection process of computing workers, their ordering and the distribution of the adequate chunk sizes remain an open problem. [17] has built up a scheduling model with the assumptions that computations are suspended by communications. Other works have been done on specialized infrastructure, such as in [21] for distributed bus network.

The rest of the paper is structured as follows. Section 2 describes the system model we consider. Our dynamic scheduling algorithm is described in section 3. In section 4, we evaluate the proposed model and discuss the obtained results. Section 5 concludes the work.

2. SYSTEM MODEL

The computing platforms that we are targeting are Grids and Clouds of computing resources. As shown in Figure 1, a computing platform consists of nodes which are connected via a network link whose speed dictates the speed of the

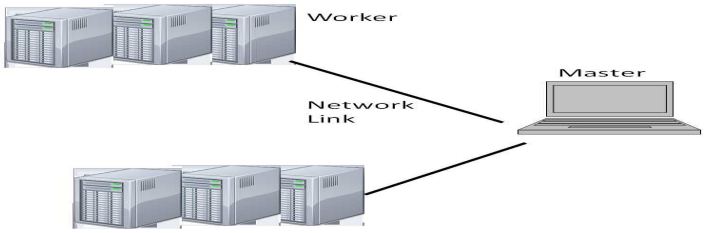


Fig. 1: Computing Platform Model

communication processing when data is transmitted among the nodes. A node consists of multiple cores. A core is the smallest processing unit available in the system. A core can be a context, a processor or a physical core.

Upon receiving of an application or a user's request, the master divides the whole application, of a total load of W_{total}, into different tasks and schedules these tasks to the selected computing workers based on their individual capability in terms of: the conditions of the communication link between the master and each computing worker, and the existing computing power of each computing worker. Each computing worker in the system has a computing capacity, μ_i, where $i = 1, ..., N$, and N is the number of computing workers in the system. In the rest of the paper, we use the terms task and chunk interchangeably.

Consider a portion of the total load, $chunk_i \leq W_{total}$, which is to be processed on worker i. We model the time required for a computing worker to perform the computation, TP_i, as:

$$TP_i = \theta_i + \frac{chunk_i}{\mu_i} \quad (1)$$

where θ_i, is a fixed latency, in seconds, for starting the computation at the computing worker r.

We model the time for sending $chunk_i$ units of load to computing worker i, TC_i, as:

$$TC_i = \lambda_i + \frac{chunk_i}{C_i} \quad (2)$$

where the component λ_i in TC_i, is a latency, in seconds, incurred by the master to initiate data transfer to worker r. C_i is the data transfer rate in terms of units of computing loads per second that can be provided by the communication link from the master to the worker i. The λ_i component may caused by the delay when initiating communicating from the master to the worker i by using SSH or any Grid or Cloud software which is used to access the worker i.

3. SCHEDULING ALGORITHM

To obtain an optimal performance for a single round, our main objective is to determine the chunk size that is to be assigned to each computing worker so that all computing

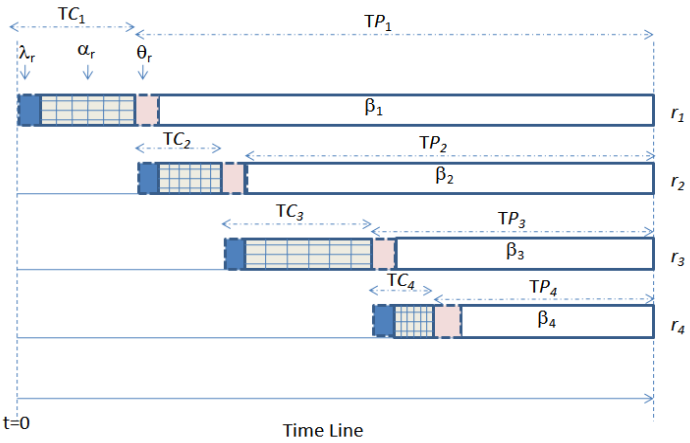


Fig. 2: Distribution of Chunk Sizes to Computing Workers in a Single Round

workers complete their computations at the same time. The total load W_{total} computation units is divided into N computing worker. We assume that the master starts to send chunks to N available computing workers in a sequential fashion. Consequently, every computing worker i will get its chunk size $chunk_i$ to process. Figure 2 shows an example, in which four computing workers are used.

Let us define $\alpha_i = \frac{chunk_i}{C_i}$, is the time required to transfer $chunk_i$ form the master to the worker i by using the communication link between the master and the computing worker C_i, and $\beta_i = \frac{chunk_i}{\mu_i}$ is the time required to compute the chunk $chunk_i$ by the computing worker i with computing processing capacity of μ_i. The time required by the first computing worker to complete the assigned task, $chunk_1$, is given by the following formula:

$$\begin{aligned} T_1 &= TC_1 + TP_1 \\ &= \lambda_1 + \frac{chunk_1}{C_1} + \theta_1 + \frac{chunk_1}{\mu_1} \\ &= \lambda_1 + \theta_1 + chunk_1 \left(\frac{1}{C_1} + \frac{1}{\mu_1} \right) \end{aligned} \quad (3)$$

The time required by the second computing worker to complete its task, $chunk_2$, is given by the following formula:

$$\begin{aligned} T_2 &= TC_1 + TC_2 + TP_2 \quad (4)\\ &= \lambda_1 + \frac{chunk_1}{C_1} + \lambda_2 \\ &\quad + \frac{chunk_2}{C_2} + \theta_2 + \frac{chunk_2}{\mu_2} \\ &= \lambda_1 + \frac{chunk_1}{C_1} + \lambda_2 + \theta_2 + chunk_2\left(\frac{1}{C_2} + \frac{1}{\mu_2}\right) \end{aligned}$$

In general, the time required by the computing worker i to complete its task, $chunk_i$, is given by:

$$\begin{aligned} T_i &= \sum_{j=1}^{j=i-1} TC_j + TC_i + TP_i \qquad i = 1, ..., N \quad (5)\\ &= \sum_{j=1}^{j=i-1} (\lambda_j + \frac{chunk_j}{C_j}) + \theta_i + chunk_i\left(\frac{1}{C_i} + \frac{1}{\mu_i}\right) \end{aligned}$$

The time required by the last computing worker N to complete the task, $chunk_N$, is given by:

$$\begin{aligned} T_N &= \sum_{j=1}^{j=N-1} TC_j + TC_N + TP_N \quad (6)\\ &= \sum_{j=1}^{j=N-1} (\lambda_j + \frac{chunk_j}{C_j}) + \theta_N + chunk_N\left(\frac{1}{C_N} + \frac{1}{\mu_N}\right) \end{aligned}$$

To achieve the goal that all workers complete the computation of their assigned chunks at the same time, it is required that:

$$T_1 = T_2 = ... = T_N = T \quad (7)$$

To simplify the formulas, we assume that:

$$\lambda_1 = \lambda_2 = ... = \lambda_N \quad (8)$$

and

$$\theta_1 = \theta_2 = ... = \theta_N \quad (9)$$

Then based on the Equation 7 and the Equations 3, 4, 5, and 6, we obtain:

$$\begin{aligned} \frac{chunk_1}{\mu_1} &= \lambda + chunk_2\left(\frac{1}{C_2} + \frac{1}{\mu_2}\right) \quad (10)\\ \frac{chunk_2}{\mu_2} &= \lambda + chunk_3\left(\frac{1}{C_3} + \frac{1}{\mu_3}\right) \quad (11)\\ &... \\ \frac{chunk_{i-1}}{\mu_{i-1}} &= \lambda + chunk_i\left(\frac{1}{C_i} + \frac{1}{\mu_i}\right) \quad (12)\\ &... \\ \frac{chunk_{N-1}}{\mu_{N-1}} &= \lambda + chunk_N\left(\frac{1}{C_N} + \frac{1}{\mu_N}\right) \quad (13) \end{aligned}$$

Based on Equation 10, we obtain the following equation:

$$chunk_2 = \frac{C_2\mu_2 chunk_1}{\mu_1(C_2+\mu_2)} - \frac{C_2\mu_2\lambda}{C_2+\mu_2} \quad (14)$$

Based on Equations 11 and 14, we obtain the following formula:

$$\begin{aligned} chunk_3 &= \frac{C_3\mu_3 chunk_2}{\mu_2(C_3+\mu_3)} - \frac{C_3\mu_3\lambda}{C_3+\mu_3} \quad (15)\\ &= \frac{C_2C_3\mu_3 chunk_1}{\mu_1(C_2+\mu_2)(C_3+\mu_3)} \\ &\quad - \frac{C_2C_3\mu_3\lambda(1+\frac{C_2+\mu_2}{C_2})}{(C_2+\mu_2)(C_3+\mu_3)} \quad (16) \end{aligned}$$

Based on Equation 12, and by substitutions, for a computing worker i, we obtain the following formula:

$$\begin{aligned} chunk_i &= \frac{\prod_{j=2}^{j=i} C_j\mu_i chunk1}{\mu_1\prod_{j=2}^{j=i}(C_j+\mu_j)} \\ &\quad - \lambda\frac{\prod_{j=2}^{j=i} C_j\mu_i(1+\sum_{k=2}^{k=j}\frac{\prod_{k=2}^{k=j}(C_k+\mu_k)}{\prod_{k=2}^{k=j} C_k})}{\prod_{j=2}^{j=i}(C_j+\mu_j)} \quad (17) \end{aligned}$$

As $W_{total} = \sum_{i=1}^{i=N} chunk_i$, and based on Equation 17, then we obtain the value of $chunk_1$ as follows:

$$\begin{aligned} chunk1 &= \\ &\frac{W_{total} + \lambda(\sum_{i=2}^{i=N}\frac{\prod_{i=2}^{i=N} C_j\mu_i(1+\sum_{k=2}^{k=i-1}\frac{\prod_{k=2}^{k=i}(C_k+\mu_k)}{\prod_{k=2}^{k=i} C_k})}{\prod_{j=2}^{j=i}(C_j+\mu_j)})}{1+\sum_{i=2}^{i=N}\frac{\prod_{j=2}^{j=i} C_j\mu_i}{\mu_1\prod_{j=2}^{j=i}(C_j+\mu_j)}} \quad (18) \end{aligned}$$

Based on Equation 17, the remaining chunks, $chunk_i$, $i = 2, ..., N$, can be obtained by substituting the value of $chunk_1$ obtained by Equation 18.

4. PERFORMANCE EVALUATION

In this section, we evaluate the efficiency of our scheduling algorithm in different scenario. The efficiency of the algorithm is measured and compared to the *ideal* makespan. The best makespan is calculated by dividing the total workload over the summation of the computing powers of the individual workers($\frac{W_{total}}{\sum_{i=1}^{i=N}\mu_i}, i = 1, ..., N$). In particular, we analyze the impact of the system parameters (μ_i, C_i) on the performance of our scheduling algorithm.

4.1 Experimental Runs

The experiments use a set of 10 computing workers. We vary the heterogeneity degree of the workers and study the performance of the scheduling algorithm. In a first step, we study the performance of the algorithm running on homogeneous clusters with increasing network capacity.

Table 1 shows the experimental values in which all the computing workers of the cluster have similar computing powers and they are connected to the master via similar network capacity. In a second step, we run the experiments with heterogeneous processing powers and homogeneous network connecting the master to the computing workers. We study the performance of the algorithm with increasing network capacity. Table 2 shows computing processing powers which have been randomly generated. In these experiments, the network and the processing powers have been chosen in a way that the computation to communication ratio can represent realistic scenarios. To that purpose, we have run a parallel MPEG video compressor in our heterogeneous Lab cluster. An input video is composed of frames and each frame can be processed independently [22]. Table 4 shows the communication and the execution times of 600MByte of raw video and the ratio of communication time to execution time in our experimental environment.

In order to analyze the impact of the degree of heterogeneity of resources on the scheduling algorithm, experiments are conducted using clusters of different heterogeneity. We randomly generate computing and communication resources by variating the standard deviation and by using the same mean of 0.50 and 30 for computation and communication respectively. Table 3 shows the standard deviations for generating the different set of communication and computation respectively. For every run, we compute a normalized makespan relative to the *ideal* makespan. Every run is executed for 100 times and the average over the runs is taken.

Table 1: Experimental Runs using Homogeneous Resources ($N = 10, W_{total} = 2000, \mu_i = 1, \lambda = \theta = 1$)

Run	Value of $C_i (i = 1, ..., N)$
1	10
2	20
3	30
4	40
5	50

Table 2: Experimental Runs using Homogeneous Resources ($N = 10, W_{total} = 2000, \mu_i = 0.45, 0.65, 0.84, 0.5, 0.4, 0.44, 0.37, 0.55, 0.64, 0.4, \lambda = \theta = 1$)

Run	Value of $C_i (i = 1, ..., N)$
1	10
2	20
3	30
4	40
5	50

Table 3: Heterogeneity Change for Communication and Computation in Experimental Scenarios

Standard Deviation for μ	Standard Deviation for C
0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5	0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5

Table 4: Video Compressor Communication and Computation Times

Transfer Time (Sec)	Computation Time (Sec)	Ratio of Communication to Computation
26.375	578.9375	21.95023697

4.2 Performance Analysis

In devising a scheduling algorithm for a single-round-based applications, several requirements have to be considered. Those requirements considerations have an impact on the experimental results:

a) Minimize waiting time of processes. This waiting time can be generated in 2 cases:
   - Processes wait for other processes to complete: after being assigned their chunks, processes which complete their executions first will wait for other processes to complete their execution. Our scheduling algorithm imposes that all computing workers complete their executions at the same time.
   - Processes wait for other processes to start: as our algorithm distributes the chunks to the processes in a sequential fashion, processes wait to receive their chunks. A selection policy of resources can decrease the impact of this problem on the overall performance of the scheduler. Our algorithm does not impose a selection policy and any selection policy can be used.

Figure 3 shows the relative performance of our scheduling algorithm on a homogeneous environment. The makespan decreases when the network capacities increase to become close to the ideal makespan with large network capacities; i.e., C=60. Figure 4 shows the chunk sizes that are distributed to the computing workers according to the scheduling algorithm. It shows that as the network capacity increases between the master and the computing worker, the chunk size decreases. This is to accommodate quicker transfer time for a current process during which previous processes are being executed. The scheduling algorithm shows a good performance with heterogeneous computing powers as shown in Figure 5. Figure 6 shows the corresponding chunk sizes distribution. To know the performance of the

scheduling in a heterogeneous environment for both computing and networking resources, we calculate the relative makespan in different heterogeneous sets of computing and networking scenarios as shown in Figure 7. The scheduling algorithm achieves a relative makespan of 1.21 for low processing heterogeneity (standard deviation of 0.1) and medium networking heterogeneity (standard deviation of 6). The performance of the algorithm decrease with high computing heterogeneity (standard deviation of 0.35) and high network heterogeneity (standard deviation of 10).

At the performance level, while it looks intuitive that a selection policy by increasing of networking capacities will achieve a better performance than using no selection, as the waiting time of the remaining processes waiting for the chunk to be distributed to the previous processes will be less, yet our first experiments were to schedule loads without any selection policy. By sorting the resources based on their computing capacities of resources in an increasing order, the scheduling algorithm achieves a minimum makespan of 1.20 and a maximum is 1.31, as shown in Figure 8. When resources are sorted by network capacities the algorithm achieves a relative performance of a minimum of 1.19 and a maximum of 1.22, as shown in Figure 9; an improvement over the performance of the algorithm with no selection policy or with computing-based selection.

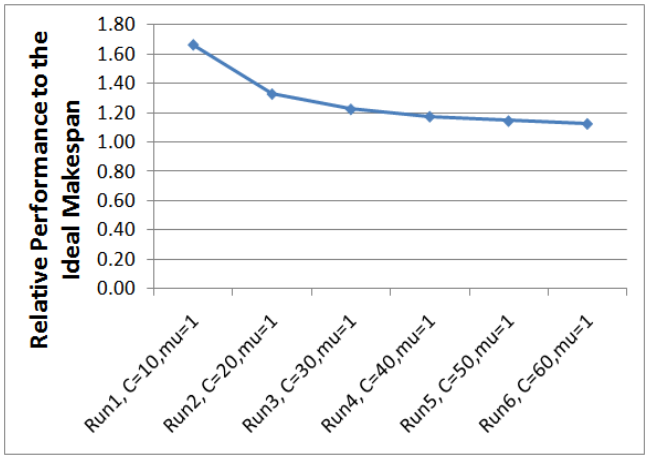


Fig. 3: Relative Performance of the Scheduling Algorithm on Homogeneous Clusters with Increasing Network Capacities.

5. CONCLUSION

In this paper, we presented a dynamic scheduling algorithm for divisible load applications to a set of heterogeneous computing workers to obtain an optimal performance. The algorithm uses single round, as this is the most used design algorithm by developers for its simplicity to program. Taking into consideration the processing and the networking capacities, our algorithm shows a good performance compared

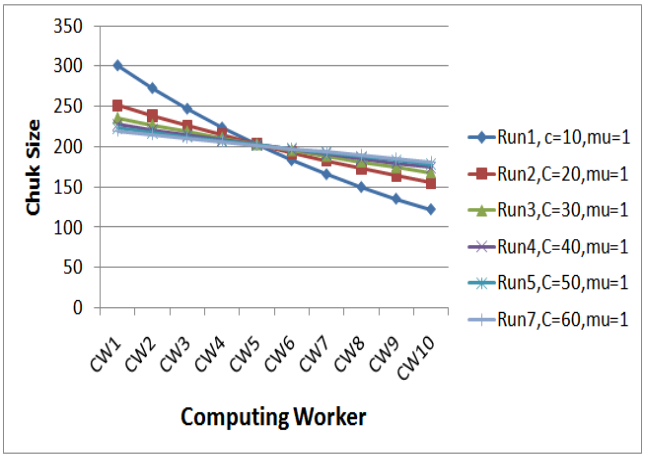


Fig. 4: Chunks Allocated to Computing Workers on Homogeneous Clusters with Increasing Network Capacity.

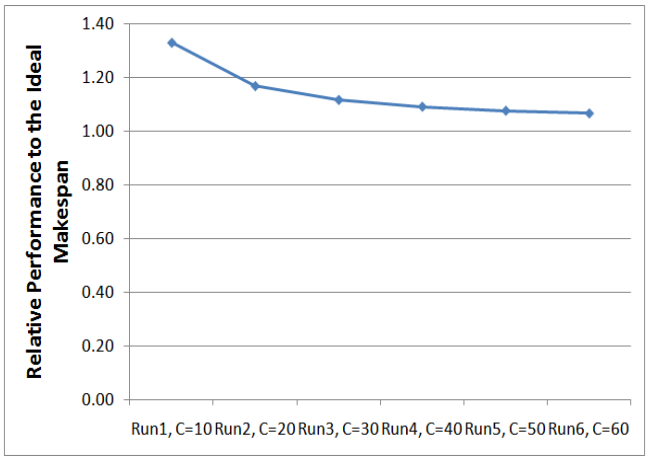


Fig. 5: Relative Performance of the Scheduling Algorithm with Heterogeneous Computing Powers and No Selection Policy versus Increasing Network Capacity.

to the ideal performance, in particular whenever a selection policy is applied. The performance results show that our algorithm has a relative makespan of 1.19 with increasing network and processing heterogeneity of of 9 and 0.3 respectively, by using a network-based selection with increasing order of network capacities.

References

[1] Ian Foster. ŤWhat is the Grid? A Three Point ChecklistŤ. Argonne National Laboratory and University of ChicagoŤ. 20 July 2002

[2] Ian Foster and Carl Kesselman. ŤThe Grid: Blueprint for a Future Computing InfrastructureŤ. Morgan Kaufmann, 1998

[3] R. Buyya, C.S. Yeo, and S. Venugopal, Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities, Keynote Paper, in Proc. 10th IEEE International Conference on High Performance Computing and Communications (HPCC 2008), IEEE CS Press, Sept. 25Ú27, 2008, Dalian, China

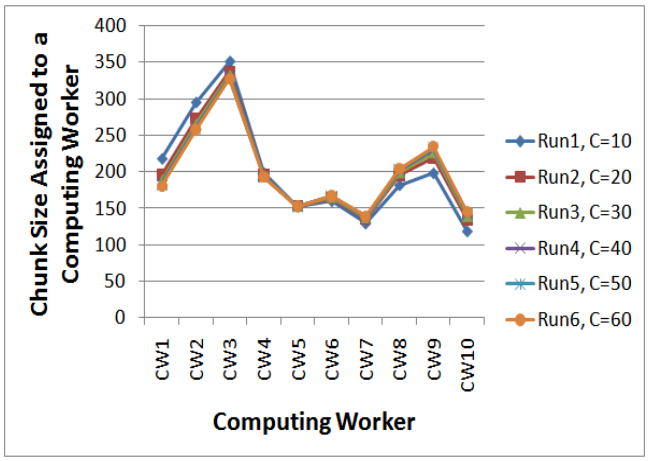


Fig. 6: Chunk Sizes Assigned to Heterogeneous Computing Powers with No Selection Policy versus Increasing Network Capacity.

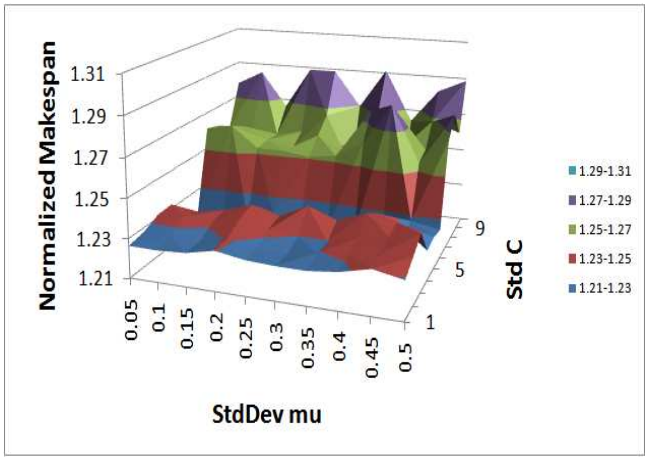


Fig. 7: Performance of the Scheduling Algorithm with Increasing Platform Heterogeneity and No Selection Policy.

[4] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems. Volume 25, Issue 6, June 2009

[5] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia. Above the Clouds: A Berkeley View of Cloud computing. Technical Report No. UCB/EECS- 2009-28, University of California at Berkley, USA, February 10, 2009

[6] V. Bharadwaj, D. Ghose, and T. Robertazzi. Divisible load theory: a new paradigm for load scheduling in distributed systems. Cluster Computing, 6(1):7Ű17, 2003

[7] D. Ghose and T. Robertazzi, editors, "Special issue on Divisible Load Scheduling", Cluster Computing, 6, 1, 2003

[8] Maciej Drozdowski and Powel Wolniewicz, "Experiments with Scheduling Divisible Tasks in Cluster of Workstations", Euro-Par 2000, pp.311-319, 2000

[9] D. Altilar and Y. Paker, "An optimal Scheduling Algorithm for Parallel Video Processing", In IEEE Int. Conference on Multimedia Computing and Systems, IEEE Computer Society Press, 1998

[10] D. Altilar and Y. Paker, "Optimal Scheduling Algorithms for Communication Constrained Parallel Processing", In Euro-Par 2002, LNCS 2400, pp. 197-206, Springer Verlag, 2002

[11] Leila Ismail and Driss Guerchi, "Performance Evaluation of Convolution on the IBM Cell Processor", IEEE Transactions on Parallel and Distributed Systems, 01 Apr. 2010, IEEE computer Society Digital Library. IEEE Computer Society, http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.70

[12] C. Lee and M. Hamdi, "Parallel Image Processing Applications on a Network of Workstations", Parallel Computing, Vol. 21, pp. 137-160, 1995

[13] V. Bharadwaj and S. Ranganath, "Theoretical and Experimental Study on Large Size Image Processing Applications Using Divisible Load Paradigms on Distributed Bus Networks", Image and Vision Computing, Vol.20, nos.13-14,pp.917-1034, 2002

[14] Ian Foster, "Designing and Building Parallel Programs", Addison-Wesley (ISBN 9780201575941), 1995

[15] V. Bharadwaj, D. Ghose, and V. Mani, "Multi-Installment Load Distribution in Tree Networks with Delays", IEEE Transactions Aerospace and Electronics Systems, vol.31, no.2, pp.555-567, 1995

[16] Yang Yang, Krijn van der Raadt, and Henri Casanova, "Multiround Algorithms for Scheduling Divisble Loads", IEEE Transactions on Parallel and Distributed Systems, vol. 16, no.11, November 2005

[17] Maciej Drozdowski and Marcin Lawenda, "Multi-installment Divisi-

ble Load Processing in Heterogeneous Systems with Limited Memory", PPAM 2005, LNCS 3911, pp.847-854, 2006

[18] Olivier Beaumount, Henri Casanova, Arnaud Legrand, Yves Robert, Yang Yang, "Scheduling Divisible Loads on Star and Tree Networks: Results and Open Problems", IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 3, pp.207-218, March 2005

[19] Leila Ismail, Bruce Mills, and Alain Hennebelle, "A Formal Model of Dynamic Resource Allocation in Grid Computing Environment", Proceedings of The IEEE Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD2008), Phuket, Thailand, pp. 685-693, 2008

[20] Yang Yang, Henri Casanova, Maciej Drozdowski, Marcin Lawenda, Arnaud Legran, "On the Complexity of the Multi-Round Divisible Load Scheduling", Report No. 6096, ISSN0249-6399, INRIA, January 2007

[21] S. Bataineh, T.-Y. Hsiung, and T.G. Robertazzi, ŞClosed Form Solutions for Bus and Tree Networks of Processors Load Sharing a Divisible Job,Ť IEEE Trans. Computers, vol. 43, no. 10, Oct. 1994.

[22] K. Shen, L.A. Rowe, and E.J. Delp, ŞA Parallel Implementation of an MPEG1 Encoder: Faster than Real-Time!,Ť Proc. SPIE Conf. Digital Video Compression: Algorithms and Technologies, pp. 407-418, Feb. 1995.

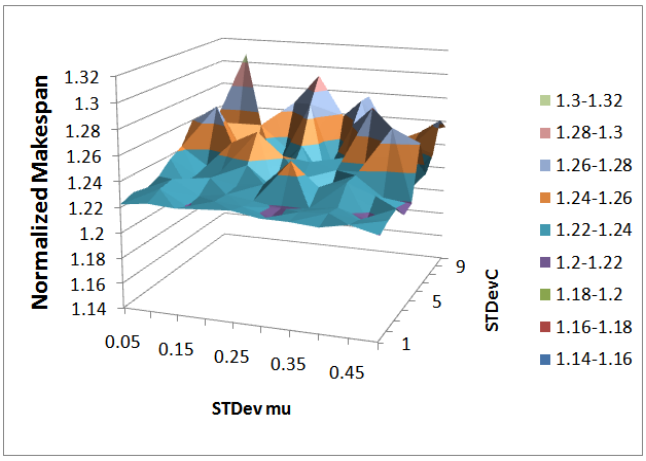


Fig. 8: Performance of the Scheduling Algorithm with Increasing Platform Heterogeneity and Computing-based Selection Policy.

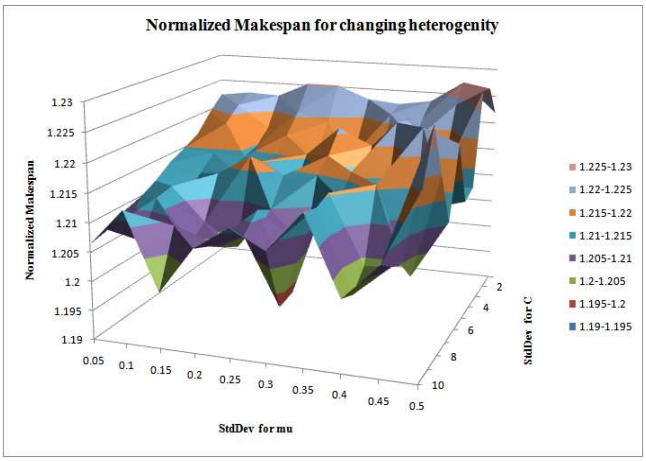


Fig. 9: Performance of the Scheduling Algorithm with Increasing Platform Heterogeneity and Network-based Selection Policy.

A Prior-knowledge-based Parallel Job Scheduling Strategy for Cluster-based Processing System of Remote Sensing Image

Yan Ma [*1,2,3], Dingsheng Liu [1], Canen Tang [1,3]

[1]Center for Earth Observation and Digital Earth, Chinese Academy of Sciences(CAS)
No.9 Dengzhuang South Road, Haidian District, Beijing 100094, China
[2]Institute of Electronics, CAS. No. 19 Beisihuan Xilu, Beijing 100190, China
[3] Graduate University of Chinese Academy of Sciences
{dsliu, yanma}@ceode.ac.cn

Abstract. Effective job scheduling schemes play a critically important role in cluster-based parallel processing system for remote sensing image. However, the recent job scheduling strategies without accurate estimated run time of algorithms commonly treat various parallel algorithms as undifferentiated jobs, which could cause blindness in resource allocation, and always lead to low system utilization and long average weighted turnaround time of job. To properly settle the problems above, APKB an Adaptive Prior-Knowledge-Based parallel job scheduling strategy is proposed in this paper. The length (average run time of processing per unit data) of various parallel algorithms that recorded in the prior knowledge database with self-learning ability are used for estimating the accurate run time of parallel jobs. With the accurate estimates of job run time and system load that dynamically computed by fuzzy model, the optimized resource allocation policy will be used in APKB strategy to shorten the average weighted turnaround time of job, and finally to achieve the load balancing of the entire system. Through experimental and comparative analysis, its outstanding scheduling efficiency is showed in this paper.

Key words. Parallel Processing System for Remote Sensing Image; Dynamic Scheduling Strategy; Load Balancing

1 Introduction

Remote sensing image processing is a compute-intensive and time-consuming task. And the increasing image scales pose many computational challenges for cluster-based processing system for remote sensing image. A suitable parallel job scheduling strategy is the key to achieving high parallel efficiency [1-4]. Commonly, the Operation and Management sub System (OMS) which responsible for parallel job scheduling always uses some cluster scheduling strategies like FIFO, First-Fit, Greedy, Priority, Weighted round-robin, Backfilling and so on. Due to the lack of accurate estimated run time of jobs and real-time load of system, those scheduling approaches which simply assigned to different jobs with the same number of computing resources, always suffered from poor system utilization and long average weighted turnaround time of job.

To properly solve the issue above, APKB an Adaptive Prior-Knowledge-Based parallel job scheduling strategy is proposed in this article. In this strategy, the prior knowledge database is imposed for accurately estimating the run time of parallel jobs. And with the accurate estimates of job run time and system load gained dynamically, the system utilization and performance could be greatly improved through proper allocation of system resources during job scheduling.

2 Adaptive Prior-Knowledge-Based Scheduling Strategy

Adaptive Prior-Knowledge-Based scheduling strategy is a dynamic parallel job scheduling scheme for cluster-based parallel remote sensing image processing system. By using adaptive resource allocation policy, APKB strategy dynamically allocates the system resources to various parallel jobs that enter the system according to the accurate estimates of job run time and current system load. Rather than the simple even allocation policy used by other scheduling strategies, this adaptive resource allocation policy would automatically adapt to the changing system load conditions and randomly arrived jobs. The mechanism of APKB strategy is illustrated as follows.

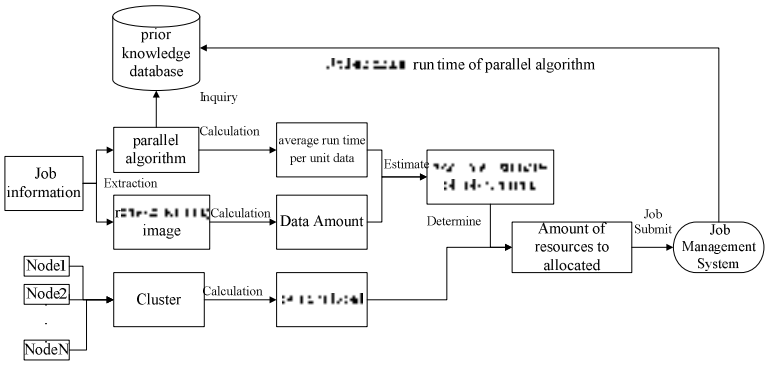


Fig. 1. The mechanism of APKB strategy

The mechanism of APKB strategy is implemented under following steps:

1) Estimates the accurate run time of job. By querying prior knowledge database with algorithm name of job, the relevant length (average run time of processing per unit data) of that parallel algorithm will be get to calculate the average run time per unit data with single node. Accordingly, this average run time together with the data amount of image to be processed are used for estimating the accurate run time of job.

2) Gain system load of the cluster. Through continuously monitoring of the load conditions of all nodes, like CPU utilization, Memory utilization, I/O speed of disk and response time, the load of each computing node would be dynamically gained. And the system load of the cluster is the weighted value of the node load.

3) Implement adaptive resource allocation policy. Dynamically allocates the system resources to the arriving parallel jobs according to the accurate estimates of job run time and current system load.

4) Job submission. Finally the jobs submit to cluster scheduler for execution. Update the actual run time of the algorithm into the prior knowledge database when finishing job execution.

2.1 The Accurate Estimation of Job Run Time

The accurate estimates of job run time is determined by T_{ave} which is the average run time of processing per unit data with single computing node and m which represented for the total data amount of remote sensing image to be processed. When arrives a new job request, the value of T_{ave} would be queried from prior knowledge database by the APKB strategy. And obviously the accurate estimates of job rum time would be equals to $T = T_{ave} \times m$.

The prior knowledge database is used for storing the basic information of various parallel algorithms, like T_{ave} and N_{tal} which is the total number of algorithm runs. Whenever the task is completed, the value of T_{ave} and N_{tal} in prior knowledge database should be updated.

The new value of T_{ave} is: $$T_{ave} = (T_{ave} \times N_{tal} + t_r / (n * m)) / (N_{tal} + 1) \quad (2.1)$$

Assume that the actual job run time is t_r when using m assigned computing nodes and dealing with image data of m MB.

2.2 Estimating System Load Using Fuzzy Membership Functions

The system load which considered as a synthesized metrics of whole cluster is closely related to CPU utilization, memory Utilization, disk I/O speed and system responding time. Usually, it is hard to exactly tell whether the utilization of CPU or memory is high or low, as the problem its self have some kind of uncertainties. Therefore, three simplified fuzzy membership functions fL(u), fM(u) and fH(u) depicted in figure 2 are put forward to respectively define three levels High, Middle and Low.

According to fuzzy statistical method, the expression of those three simplified fuzzy membership functions [5-7] put forward is as follows, where u is the utilization of CPU.

$$fL(u) = \begin{cases} 1 & 0 \le u \le 0.15 \\ 1.6 - 4u & 0.15 < u \le 0.4 \\ 0 & 0.4 < u \le 1 \end{cases} \qquad fH(u) = \begin{cases} 0 & 0 \le u \le 0.6 \\ 4u - 2.4 & 0.6 < u < 0.85 \\ 1 & 0.85 \le u \le 1 \end{cases} \quad (2.2)$$

$$fM(u)=\begin{cases}1 & 0\le u\le 0.3\\ \frac{20}{3}u-2 & 0.3<u<0.45\\ 1 & 0.45\le u\le 0.55\\ \frac{14}{3}-\frac{20}{3}u & 0.55\le u\le 0.7\\ 0 & 0.7\le u\le 1\end{cases} \tag{2.3}$$

When given the CPU utilization u of any time, the CPU utilization level could be pointed out as L, M or H according the above fuzzy membership functions. Similarly, when considering four factors CPU utilization u_1, memory utilization u_2, disk I/O Speed u_3 and responding time u_4, we will get four similar groups of fuzzy membership functions. While define a factor Set U={ u_1,u_2,u_3,u_4 } , then there exists a fuzzy relationship from factor set to decision-making set V={L,M,H}. This fuzzy relationship is expressed by $R_{3\times4}$, and r_{ij} indicates the level set that factor j belongs to.

Obviously, the larger $r_{1j}$is, the higher possibility that u_1 would belong to L set. Considering that the node with low CPU utilization leads to low CPU load, so the CPU load is like $r_{11}\times(-2)+r_{21}\times0+r_{31}\times2$. Similarly, the calculate method of each node load factor is showed in following table.

	CPU	Memory	I/O	Reponses
L	$r_{11}\times(-2)$	$r_{12}\times(-2)$	$r_{13}\times(-4)$	$r_{14}\times2$
M	$r_{21}\times0$	$r_{22}\times0$	$r_{23}\times0$	$r_{24}\times0$
H	$r_{31}\times2$	$r_{32}\times2$	$r_{33}\times4$	$r_{34}\times(-2)$

Table. 2.1. The calculate method of each load factor of node

It is indicated that the coefficients (2, 2, 4, 2) given in the above table can also be regarded as weight of each factor (0.2, 0.2, 0.4, 0.2) in calculating system load. Accordingly, the load of each computing node is:

$$\text{Load}=[r_{11}\times(-2)+r_{31}\times2+r_{12}\times(-2)+r_{32}\times2+r_{13}\times(-4)+r_{33}\times4+r_{14}\times2+r_{34}\times(-2)] \tag{2.4}$$

Finally, the system load of the entire cluster is Load_{ave}:

$$Load_{ave}=\sum_{i=1}^{N}\omega_i Load_i \tag{2.5}$$

Assume that there are N computing nodes in the cluster, and the load of the ith node is $Load_i$, ω_i is the weight of each node and $\sum_{i=1}^{N}\omega_i=1$. The nodes with high performance will have a high weight value ω_i.

2.2 Adaptive Resource Allocation Policy

The main ideal of the adaptive resource allocation policy is: the parallel algorithms are classified into four different categories according to T_{ave} the average run time when processing per unit data with single node. Also, the job queue is divided into four separated sub queue by the amount of resources to be allocated. The mapping relationship from algorithm categories to job queue is as follows.

$$N=\begin{cases}2, & (0<T\le800)\\ 4,(800<T\le1600)\\ 6,(1600<T\le2400)\\ 8,(2400<T)\end{cases} \tag{2.8}$$

Furthermore, Load_{ave} the real-time system loads of the whole cluster are classified to four levels, each level have its Corresponding index value λ, the mapping relationship between λ and Load_{ave} is expressed as follows.

$$\lambda=\begin{cases}0.5,(Load_{ave}>15)\\ 1,(10<Load_{ave}\le15)\\ 1.5,(5<Load_{ave}\le10)\\ 2,(0<Load_{ave}\le5)\end{cases} \tag{2.9}$$

Eventually, the amount of resource to be allocated is dynamically determined by the accurate estimates of job run time and the real-time system load of cluster. And the accurate estimates of job run time could be calculated by T_{ave} and data amount. Therefore, the amount of resources allocated to each job equals to $n=N\times\lambda$.

As a matter of fact, when more parallel jobs are processed in the system, the estimates of job run time will be more accurate due to the self-learning ability of prior knowledge database, and finally the resource allocation would be much more reasonable and adaptive. Just benefit from the proper resource allocation, the run time

gap between long parallel job and short parallel job is shorten, and also the time overhead caused by long parallel jobs and short parallel jobs waiting for each other is reduced. Accordingly the optimized performance of the whole system would be achieved.

3 Performance Evaluation and comparison

Through the quantitative evaluation of the average weighted turnaround time and system utilization of both APKB and FIFO strategy, their performances are comparatively analyzed.

3.1. Comparative Analysis of Average Weighted Turnaround Time

Suppose that n parallel jobs are requested randomly, $T_{process}$, T_{wait} and T_{weight} are respectively the actual run time, waiting time and average weighted turnaround time. And the $T_{\text{FIFO-process-i}}$ and $T_{\text{FIFO-wait-i}}$ are stands for the run time and waiting time of the ith jobs when using FIFO strategy, the $T_{\text{APKB-process-i}}$ and $T_{\text{APKB-wait-i}}$ are stands for the run time and waiting time of the jth jobs when using APKB strategy.

The average weighted turnaround time for FIFO strategy $T_{\text{FIFO\_average}}$ is:

$$T_{\text{FIFO\_ave}} = \sum_{i=0}^{n} \frac{(T_{FIFO\_process\_i} + T_{FIFO\_wait\_i})}{n * T_{FIFO\_process\_i}} \tag{3.1}$$

The average weighted turnaround time for APKB strategy $T_{\text{APKB\_average}}$ is:

$$T_{\text{APKB\_ave}} = \sum_{j=0}^{n} \frac{T_{\text{APKB}\_process\_j} + T_{\text{APKB}\_wait\_j}}{n * T_{\text{APKB}\_process\_j}} \tag{3.2}$$

The differences between $T_{\text{FIFO\_average}}$ and $T_{\text{APKB\_average}}$ is :

$$T_{\text{FIFO\_ave}} - T_{\text{APKB\_ave}} = \frac{1}{n}\left(\sum_{i=0}^{n} \frac{T_{FIFO\_wait\_i}}{T_{FIFO\_process\_i}} - \sum_{j=0}^{n} \frac{T_{\text{APKB}\_wait\_i}}{T_{\text{APKB}\_process\_i}}\right) \approx \frac{to}{n}\left(\sum_{i=0}^{n} \frac{1}{T_{FIFO\_process\_i}} - \sum_{j=0}^{n} \frac{1}{T_{\text{APKB}\_process\_i}}\right) \tag{3.3}$$

$$\sum_{i=0}^{n} T_{FIFO\_process\_i} = \sum_{j=0}^{n} T_{APKB\_process\_i} = Tp \tag{3.4}$$

$$\frac{1}{a} + \frac{1}{b} \geq \frac{1}{(a+b)/2} \tag{3.5}$$

As the waiting time of both two scheduling strategies $T_{FIFO\_wait\_i}$ and $T_{\text{APKB}\_wait\_i}$ are almost equivalent due to the random job requests, we assume that this waiting time equals to to. The difference between $T_{\text{FIFO\_average}}$ and $T_{\text{APKB\_average}}$ is estimated in equation 3.3. From the above formula we can infer that the when using FIFO strategy, there is a big gap between the run times of the long jobs and short jobs, but when using APKB strategy, the run time of each job is trend to be similar especially when lots of parallel jobs are requested. Suppose that the sums of processing time of each job when using these two scheduling strategies are nearly the same, which is showed in equation 3.4. And when there are lots of jobs requested, the time of $T_{\text{APKB}\_process\_i}$ will trend a mean value processing time Tp/n. So according to theory expressed in equation 3.5, the average weighted turnaround time for APKB strategy is much smaller. Obviously, when the gaps between the long parallel jobs and short parallel jobs are widen, the performance improvement will be more significant.

3.2. Comparative Analysis of System Utilization

The system utilization is the ratio of current system load and total ability of system load. When do job scheduling the load balancing between computing nodes is an effective way to raise system utilization.

The system utilization η is: $\eta = N_{busy} / N_{total} \times 100\%$ （3.6）

During a period time of T (T is big enough) which equals to $T = T_{\text{high}} + T_{\text{middle}} + T_{\text{low}}$, the system load could be divided into three phases as High load, middle load and low load.

So the system utilization when using FIFO strategy η_{T_FIFO} is:

$$\eta_{T\_FIFO} = \frac{T_{high}}{T} \times \eta_{high\_FIFO} + \frac{T_{middle}}{T} \times \eta_{middle\_FIFO} + \frac{T_{lowc}}{T} \times \eta_{low\_FIFO} \tag{3.7}$$

And the system utilization when using APKB strategy η_{T_APKB} is:

$$\eta_{T\_APKB} = \frac{T_{highc}}{T} \times \eta_{high\_APKB} + \frac{T_{middlec}}{T} \times \eta_{middle\_APKB} + \frac{T_{low}}{T} \times \eta_{low\_APKB} \quad (3.8)$$

While the system is high loaded, the FIFO strategy which allocated to each parallel job with same amount of nodes will finally leads to over loaded of some nodes, so the $\eta_{high\_FIFO} < \eta_{high\_APKB}$. When the system is phase of middle loaded, these two strategies will implement nearly the same resource allocation, so $\eta_{middle\_FIFO} \approx \eta_{middle\_APKB}$. But when system is poor loaded, the FIFO strategy will result in the idle of some nodes, thus $\eta_{low\_FIFO} < \eta_{low\_APKB}$. To draw the conclusion that APKB scheduling strategy would achieve a much better system utilization.

4 Experiment & Analysis

The experimental performance comparison of FIFO and APKB will be taken on Lenovo Deepcomp 6800 which composed of 18 nodes, each with dual Intel(R) 3.0Ghz processors. Network capability is 1000MBps. The resource amount allocated to each is 4 for FIFO.

(1) Comparative analysis of total time and average weighted turnaround time

As is showed in the figures the APKB scheduling strategy outperforms FIFO strategy with smaller total run time and average weighted turnaround time.

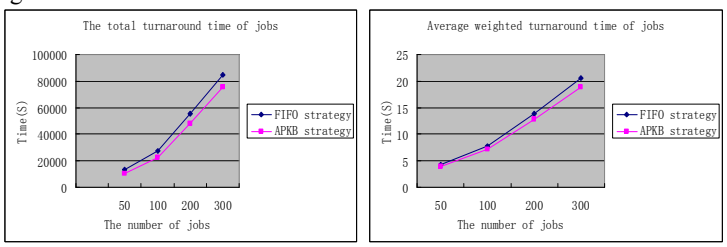
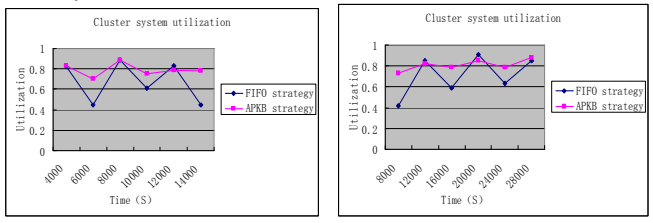
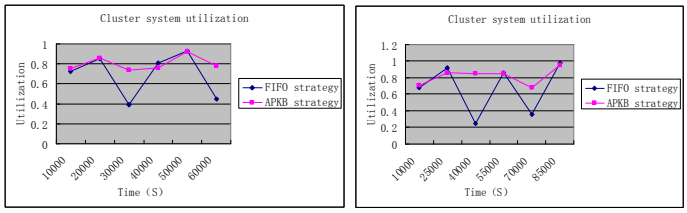


Fig. 2. Comparison of total time and average weighted turnaround time

(2) Comparative analysis of system utilization



(a) 50 samples of jobs (b) 100 samples of jobs



(c) 200 samples of jobs (d) 300 samples of jobs

Fig. 3. Comparison of system utilization

As is demonstrated in figures above, the APKB strategy gains more system utilization than FIFO. And with increasing of job samples, the system utilization trends to be stable rather than FIFO.

5 Conclusions

In this paper introduced and implemented a practical and efficient adaptive prior-knowledge-based parallel job scheduling strategy for cluster-based parallel processing system of remote sensing image. Trough accurate estimation of job run time using a prior-knowledge data base and real-time system load of cluster, an adaptive resource allocation policy is used avoid the blindness in resource allocation. Experimental results have shown that

the average weighted turnaround time of jobs is greatly reduced by the proposed scheduling strategy, and also a better system utilization and performance enhancement is achieved.

6 References

[1] George Cybenko, "Dynamic load balancing for distributed memory multiprocessors," Journal of Parallel and Distributed Computing Volume 7, Issue 2, October 1989, Pages 279-301

[2] M.H. Willebeek-LeMair, A.P. Reeves, "Strategies for Dynamic Load Balancing on Highly Parallel Computers," IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 9, pp. 979-993, Sept. 1993, doi:10.1109/71.243526

[3] Amnon Barak, Amnon Shiloh," A distributed load-balancing policy for a multicomputer," Software: Practice and Experience Volume 15 Issue 9, Pages 901 – 913 Published Online: 30 Oct 2006

[4] Jianqing Zhang, Tao Ke, Mingwei Sun, "The Parallel Computing Based on Cluster Computer in the Processing of Mass Aerial Digital Images," isip, pp.389-393, 2008 International Symposiums on Information Processing, 2008

[5] David Valencia, Antonio Plaza, Pablo Martínez, Javier Plaza, "Parallel Processing of High-Dimensional Remote Sensing Images Using Cluster Computer Architectures," IJCA, Vol. 14, No. 1, Mar 2007

[6] Li S. W., Gong J. F., Wang G. F., "Research of Workflow Scheduling Algorithm Based on Fuzzy Theory," ihmsc, vol. 1, pp.312-315, 2009 International Conference on Intelligent Human-Machine Systems and Cybernetics, 2009

[7] S. Patra, S.K. Goswami and B. Goswami," Fuzzy and simulated annealing based dynamic programming for the unit commitment problem" Expert Systems with Applications Volume 36, Issue 3, Part 1, April 2009, Pages 5081-5086

Parallel Data List Processing on Multicore-GPU Platforms

Carlos Alberto Martínez-Angeles[1], **Jorge Buenabad-Chávez**[1]
Miguel Alfonso Castro-García[2], **José Luis Quiroz-Fabián**[2]
[1]Departamento de Computación, CINVESTAV-IPN, Av. Inst. Politecnico Nal. 2508, D.F., 07360 México
Emails: jedi.knight25@gmail.com, jbuenabad@cs.cinvestav.mx
[2]Departamento de Ingeniería Eléctrica, UAM-I, Av. San Rafael Atlixco 186, D.F, 09340 México
Emails: {mcas,jlqf}@xanum.uam.mx

Abstract— *Multicore-GPU platforms are now common and affordable, yet capitalising on their parallel processing capability is not straightforward. Existing sequential and parallel software must be tuned, or designed anew, to efficiently capitalise on these platforms.*

This paper presents the design of parallel data list processing in multicore-GPU platforms, wherein application data is organised into various lists, one list for each core and GPU device, for the purpose of balancing the workload through work (data items) stealing. A novel aspect of our design is the processing of **new data dynamically generated within GPUs***. We present experimental results for three applications with different granularities and access patterns. Overall the use of GPUs can significantly improve performance, but using them profitably may not be simple.*

Keywords: Parallel Computing, Multicores, GPUs, Load Balancing, Shared Memory

1. Introduction

Multicores and GPUs are now common in laptops, desktops and compute nodes in clusters, offering unprecedented compute power to improve application performance substantially in many cases. And the cost is rather reasonable. However, not all of this compute power is readily usable. Existing software must be tuned to be able to capitalise on the parallel processing capability of such platforms. Sequential applications must be redesigned anew and implemented into a parallel multi-threaded version with shared memory communication based on semaphores, locks and/or conditional variables [1].

Many parallel application were designed for cluster computing using the message passing interface (MPI). In clusters of uniprocessor nodes, they would run in parallel with a single process running on each node. In cluster of multicore nodes, they can capitalise on the parallel capacity of these platforms *unmodified*, running as many processes for each multicore node as cores available in each node. However, processes running in the same node will communicate through message passing which, in principle, is less efficient than shared memory communication between threads because of the need to prepare, send, receive and unpack messages. Also, MPI applications must be modified to capitalise on the parallel capacity of GPUs, as using GPUs involves shared memory communication between the host platform and the GPU device [2], [3].

The Data List Management Library (DLML) is a middleware to process data lists in parallel that has undergone the above tuning in order to run more efficiently on clusters of multicore nodes with GPUs attached to some nodes. The first version of DLML was based on multiprocess parallelism and communication based on MPI message passing. *MultiCore* (MC) DLML is based on multithreaded parallelism and shared memory communication for intra-node parallelism and message passing for inter-node parallelism; it has been reported in [4], [5].

This paper presents *MultiCore-GPU* (MCG) DLML, a version able to capitalise on the parallel capacity of multicore-GPU platforms. The main issues we addressed in the design of MCG-DLML were: i) communication between the multicore platform and the attached GPU, and ii) the processing of dynamically generated data within the GPU. The last aspect is novel in that most GPU applications involve some form static processing, wherein a GPU is allocated a fixed amount of work (e.g., a scalar-vector multiplication) that is processed until completion before the GPU is allocated more work. In contrast, one of our DLML applications, the Non-Attacking Queens (NAQ) problem [6], generates new data items dynamically, which is handled as follows in MCG-DLML. Application threads running on the multicore platform insert new items back into its list. Application threads running on the GPU share an input buffer from which they read items to process and an output buffer onto which they write new items dynamically generated. Once the input buffer is exhausted, the input buffer is turned into the output buffer and the previous output buffer is turned into the input buffer.

The paper continues as follows. Section 2 presents the motivation for parallel data list processing. Section 3 outlines the organisation of MC-DLML. Section 4 presents the design of MCG-DLML. Section 5 presents our experimental platform and applications to compare the performance of MC-DLML and MCG-DLML. Section 6 presents our results. We present results for three applications with different granular-

ities and data access patterns running on a single multicore with a GPU attached, as using two or more multicore nodes would involve message-passing overhead which will become dominant. Two of our applications show much improved performance on the multicore-GPU platform; the other application shows some performance loss due to communication between the host multicore and the GPU device, but we believe it can be tuned. We conclude in Section 7.

2. Parallel Data-List Processing

In designing DLML [7], our purpose was to simplify the parallel programming of some applications through providing users with a simpler interface than that provided by MPI or Pthreads, or both. Other middlewares for parallel computing that have the same purpose are Skeletons [8] and Mapreduce [9]. In DLML, users only need to organise their data into items to *insert* into and *get* from a list using DLML functions. DLML applications run under the Single Program Multiple Data (SPMD) model: all processors run the same program but operate on distinct data lists. When a list becomes empty, it is refilled through *stealing* data items from another list transparently to the programmer. Only when `DLML_get()` does not return a data item, the calling application thread knows that no more data items are available and it can terminate. DLML functions hide synchronisation communication from users, while automatic list refilling tends to balance the workload according to the processing capacity of each processor, which is essential for good performance.

The first version of DLML [10] was designed with clusters composed of uniprocessor nodes in mind, and thus was designed based on multiprocess parallelism and message-passing communication with MPI. In this version, an application process and a DLML process are run in each node. The former runs the application code while the latter is in charge of: i) making data requests to remote nodes when the *local* list becomes empty, and ii) serving data requests from remote nodes whose list has become empty. Both tasks follow a message-passing protocol. Message passing is also used between an application process and its *sibling* (*in-same-node*) DLML process to move list items between their address spaces.

Note that the first version of DLML can run *unmodified* in clusters of multicore nodes and yet capitalise on intra-node parallelism, i.e., running in parallel on the various cores available within each multicore node. We only need to run an application process and a DLML process for each core. However, message passing would be used between DLML processes running in the same multicore node. An obvious improvement was to reduce the communication overhead of message passing between the cores in the same node, through multithreaded parallelism and communication based on shared memory.

```
void application_thread( void *my_id ) {
  DLML_dataitem item;
  DLML_list *L = DLML_thread_list( my_id );

  if ( DLML_Iam_master_thread( my_id ) ) {
    insert_data_items();
  while ( DLML_get( L, &item ))
    process_data_item( &item );
}

int main(int argc, char *argv[]) {

  DLML_init( argc, argv );
  application_thread( (void *) 0 );
  DLML_finalise();
  return 0;
}
```

Fig. 1: General organisation of an application in MC-DLML.

3. MultiCore DLML

Multicore DLML, or MC-DLML, was designed based on multi-threaded parallelism and shared memory communication for intra-node parallelism [4], [5]. Figure 1 shows a typical application in MC-DLML. The code in the `main()` procedure in Figure 1 is run by the master thread (id=0) only. The call `DLML_init()` initialises the DLML lists for each application thread, and then creates as many applications threads as cores available (by default) or as many as the user indicates as a parameter in the command running the application. The master thread then itself runs the code corresponding to application threads, `application_thread()`, passing as parameter its id 0. Within the application thread, the user code must distinguish between the master thread and all other threads: the master thread typically inserts the data items to be processed, or just the initial data items if the application dynamically generates data items, and then loops while `DLML_get()` returns a data item to process it. The user code deals with a single list, while internally several lists are implemented.

The main issues addressed in designing MC-DLML were the following: i) the locking overhead for MPI calls to be thread safe, ii) cache locality and memory consumption, and iii) intra-node synchronisation cost.

i) MC-DLML uses various application threads, usually as many as the number of cores available in each multicore node, and only one thread to make requests to remote nodes and to serve requests from remote nodes using `MPI_THREAD_FUNNELED` support, which does not involve thread-safe locking. Using one thread to make requests and another thread to serve requests simplifies coding, but requires using `MPI_THREAD_MULTIPLE` or `...SERIALIZED` level support, which involves thread-safe locking whose overhead with fine-grain applications we found to be quite high.

ii) MC-DLML uses a list for each application thread, each thread getting items from, and inserting new items

dynamically generated at, the front of the list, i.e., using lists as stacks. Hence items that have just been placed on the list are sooner removed from the list and processed while still resident in the cache. Using other list organisations, we found we were losing cache locality. We also found that, one of our applications, the non-attacking Queens (NAQ) problem, consumes much less memory when we use lists as stacks (a problem similar to traversing a tree either depth-first or breadth-first). Thus our design of MC-DLML that outperforms DLML uses a single list for each application thread, with gets from and inserts at the front. This design decision has various points in common with the "data structures in the multicore age" described in [1].

iii) Application threads running in a multicore share a workload. Although each thread uses a list to get and insert the data items, when a list becomes empty, the owner thread *steals* data items from another list. Hence the workload gets balanced according to the processing capacity of each thread. The simplest work stealing synchronisation algorithm is for a thread to lock its list every time it inserts or gets a data item from its list, and for a work stealing thread to lock its list and the lists of other threads when attempting to refill its list.

To optimise performance, we identified two *operating modes* for each thread: *normal operation* and *stealing operation*. A thread is in normal operation when it is either inserting or getting a data item. A thread is in stealing operation when it is attempting to steal data items to refill its list, or when a thread stops using its list as a result of acknowledging the synchronisation required by a stealing thread. In one of our synchronisation algorithms [5], normal operation requires no synchronisation. Each thread that inserts or gets a data item only *reads* a flag to check if a stealing operation has been signalled in order to change its operating mode. Thus in normal operation no overhead but that read is incurred. Under a stealing operation, synchronisation is needed for threads to stop using their lists so that the stealing thread does not interfere with the normal operation of the other threads. This synchronisation requires all other threads to stop using their lists in order for the stealing thread to choose the largest list to refill its list. We call this algorithm *Global Locking* (GL). Choosing the largest list is in principle a good policy, as it should tend to reduce the number of work stealing operations and the corresponding overhead.

In another algorithm we designed [4], we identify safe phases to access the lists of other threads with no or little synchronisation, thereby reducing the overhead of stealing operations. The purpose was to avoid potentially long idle periods running coarse grain applications. We call this algorithm *Low-synchronisation Locking* (LSL). LSL requires each thread to lock its list every time it enters `DLML_get()` or `DLML_insert()`, and unlock it when it returns from those procedures. The locking and unlocking consists of turning on and off a flag with atomic operations — not a mutex lock which is more costly. Then, a stealing thread only attempts to steal data items from a list that is not currently locked.

```
01 int DLML_get( LIST *L, ITEM *item ) {
02   synchronise if needed      // SYNCHRONISE
03   if  L->size > 0
04     *item = L->first         // GET ITEM
05     return 1
06   else                       // REFILL LIST
07     signal synchronisation for
08       exclusive access to other
09       list/s
10     if data items available
11       steal some
12       return 1
13     else
14       return 0
15 }
```

Fig. 2: DLML_get() main phases.

For different workloads, each of the algorithms above can be superior. GL will tend to perform well with fine-grain applications, while LSL with coarse-grain applications. Figure 2 shows the main phases of DLML_get() under either GL or LSL.

4. MultiCore-GPU MCG-DLML

4.1 Introduction

Recall that MPI parallel applications can run unmodified on multicore-node clusters, although not as efficiently as they would run if they were tuned to use multithreaded parallelism and shared memory communication. On GPUs, however, an application cannot run unless it is adapted to capitalise on the parallelism that GPUs offer.

GPUs are designed to perform well in compute-intensive, highly parallel computation typical of graphics rendering and many scientific applications that model physical phenomena. Their design devotes more transistors to data processing than to data caching and flow control [11]; the opposite is the case in general purpose processors. GPUs can process many data elements in parallel using the same program. However, this *same program* is not quite the same as that in the term Single Program Multiple Data (SPMD). In GPUs, the *same program* is synchronised by hardware, while in SPMD it is synchronised through message passing and/or shared memory synchronisation primitives specified by the programmer.

In GPUs, the *same program* is scheduled as follows. First, a thread in the host platform (e.g., a multicore) copies the data to be processed from host memory onto GPU memory, and then *invokes* the GPU threads to run the *same program* to process the data. When the GPU finishes processing the data, it signals the host thread, which may again schedule new work. Below, we describe the main components of MCG-

DLML related to processing data items in GPUs: GPU work scheduling and GPU dynamic data processing.

4.2 GPU work scheduling

Scheduling work onto a GPU consists of first copying the data to be processed from host memory onto GPU memory, and then invoking the GPU threads. This is as follows in MCG-DLML. Recall that MC-DLML uses, in each multicore node, various application threads, usually as many as the number of cores available in each multicore node, and one thread to make data requests to remote nodes and to serve requests from remote nodes. In addition to those threads, MCG-DLML uses a Host-GPU (HG) thread to move data from host memory onto GPU memory and vice versa, and multiple GPU threads in charge of processing data in GPU memory.

The HG thread is similar to the application thread shown in Figure 1 in that it repeatedly calls DLML_get() to get items while items are available. The HG thread runs on the host and the items it gets it copies them to GPU memory as described shortly. As DLML_get() steals data items from other lists if need be, the workload gets balanced among the cores and the attached GPU/s.

Getting and copying items to GPU memory is different for each of our applications in order to improve performance. Our applications include the NAQs problem, a matrix multiplication (MM) algorithm, and an image segmentation (IS) algorithm. We describe each application in detail in Section 5, and here only the general aspects of their processing in cores and the GPU. For NAQs and MM, the HG thread calls DLML_get() a number of times, say N, storing the obtained data items contiguously in host memory without processing them. It then copies the N data items to GPU memory and invokes the GPU threads to process the items.

For IS, the HG thread gets and copies to GPU memory only one item which is an image. GPU threads are then invoked to process each a pixel of the image. Each image is a matrix of size 217 rows and 181 columns, and thus 39277 GPU threads are invoked at once. For MM and NAQ, the HG thread gets and copies to GPU memory 5000 and 500 items respectively; it then invokes that many GPU threads to process each an item. The items are described in Section 5.

Invoking GPU threads is asynchronous: the HG thread can do something else immediately after invoking the GPU threads, but in our current implementation of MCG-DLML we do not capitalise on this. Copying data from GPU memory to host memory (e.g., some totals) is synchronous, however; the HG thread must wait for its completion. Moreover, a copy from GPU memory to host memory can only take place until the GPU has finished processing the last scheduled work.

4.3 GPU dynamic data processing

The NAQs problem dynamically generates new data items that need GPU memory to hold them. Hence, not all of the GPU memory is used to hold items copied from host memory. A portion of GPU memory is reserved for new items dynamically generated. Thus GPU memory is logically divided into two portions. A portion is used as input buffer to hold items copied from host memory by the HG thread; the other portion is used to hold items dynamically generated (out of processing items in the input buffer). Reading items from the input buffer and writing new items to the output buffer occur concurrently. When the input buffer is exhausted, the input buffer is turned into the output buffer and the previous output buffer is turned into the input buffer without synchronising with the host thread.

(As MM does not generate new data, we copy as many items to GPU memory as possible. In the case of IS, we try to process each image as soon as possible.)

Recall that items to copy to GPU memory are first stored contiguously in host memory. When copied into the GPU input buffer, items are also stored contiguously. Also, a pointer variable is set to point to the beginning of the output buffer. Then, all items in the input buffer are processed simultaneously: as many GPU threads as the number of items in the input buffer are run concurrently, each GPU thread accessing and processing an item whose location in the input buffer is computed based on the GPU thread ID.

When processing an item, if a GPU thread generates a new item, such thread *atomically increases* the pointer variable pointing to the output buffer, and stores the new item in the address returned by the atomic operation (i.e., the address of the previous location).

Once the input buffer is processed, the number of new items generated, say NEW, is computed from the value of the pointer variable to the output buffer. The previous input buffer is turned into the output buffer, and the previous output buffer into the input buffer, and NEW GPU threads are run to process the new input buffer. Turning input/output buffers into the other continues until no new items are generated and control is given back to the HG thread in the host platform.

Note that the processing of an item can generate various new items, but only one for each processing of an input buffer. If an item can still potentially generate new items (not all queens have been placed), it is sent back to host memory, inserting it into the list of the HG thread with DLML_insert().

5. Experimental Platform and Applications

This section describes the experimental platform and applications we used to compare the performance of both MC-DLML and MCG-DLML. We will present results for a

single multicore with a GPU attached, as using two or more multicore nodes would involve message-passing overhead which will become dominant. ***MULTICORE specification***: Intel(R) Core(TM) 2 Quad CPU Q6600 (4 cores in total) at 2.40GHz, 4 x 32KB L1 instruction caches, 4 x 32KB L1 data caches, 2 x 4MB L2 caches (each cache shared between 2 cores), and 6GB DRAM. ***GPU specification***: GeForce GTX 460, 1.53 GHz 336 CUDA Cores (7 Multiprocessors x 48 CUDA Cores/MPK), 1023 MB, 512MB L2 cache.

SOFTWARE used: CentOS release 5.7, Open MPI 1.4.4, gcc version 4.1.2, NVIDIA Corporation Cuda compilation tools, release 4.0, V0.2.1221, CUDA Driver Version/Runtime Version 4.10/4.0, CUDA Capability Major/Minor version number: 2.1. The work stealing algorithm used by both MC-DLML and MCG-DLML is the Low-synchronisation algorithm reported in [4].

Our applications include: i) image segmentation (IS) using the Mean-Shift (MSH) method, ii) a matrix multiplication (MM) algorithm, and iii) the Non-Attacking N-Queens (NAQ) problem. Image segmentation (IS) with the Mean-Shift method (MSH) is used to reconstruct 3D brain images [12], [13], through applying such method to each pixel on several 2D images (cuts). The method is expensive for the large number of cuts and high resolution required. The number of cuts is fixed, but the processing cost of each cut varies according to MSH cost, which depends on the intensity of pixels of each cut. In the DLML version, each list item is an integer value that identifies an image cut; our experiments processed 165 image cuts.

Matrix multiplication (MM), $C = A \times B$, is a *static* application because all data is known in advance. In the DLML version, each list item contains all the data to compute an element in the results matrix C, i.e.: a full row of A, a full column of B, and the x, y position of the element in C. A, B and C are $N \times N$ matrices, and we experimented with $N = 500$ and 600.

The NAQ problem consists of finding all possible ways of placing N queens on an N×N chessboard, so that no queen attacks another queen [6]. The search space of NAQ can be modelled with an N-degree search tree. The solutions are found exploring the search tree for possible solutions, eliminating those that can not be solved. In the DLML version, each list item contains a possible solution to explore formed by the number of queens to be placed and an array of size N with the position of the queens placed so far. The list *dynamically* increases and decreases as new possible solutions are generated and failed solutions are eliminated. The cost of finding a solution is a function of the depth of the relevant item in the search tree.

6. Results

This section presents experimental results on the performance of MC-DLML and MCG-DLML running our applications. In the figures shown below, the Y axis shows response time and the X axis the number of (application) threads used to run an application code. The results corresponding to MCG-DLML are labelled with the prefix *gpu-*. In the results of gpu- runs, the number of application threads shown, say N, means that $N - 1$ application threads were run in a core in the host multicore platform, while 1 thread was the HG thread responsible for scheduling work to the GPU as described in Section 4. The HG thread does not process application data; it only schedules work to the GPU.

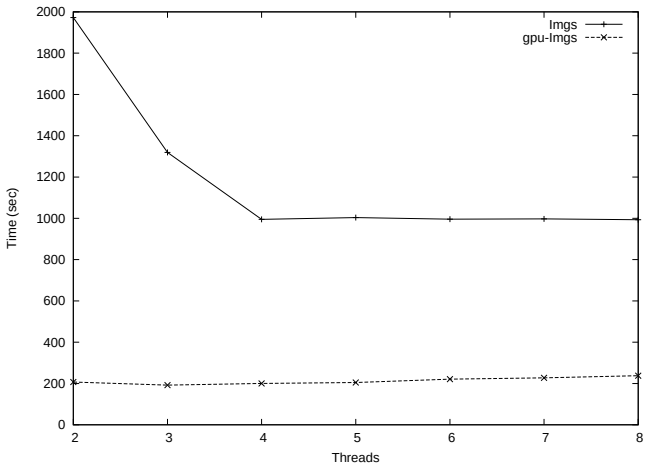


Fig. 3: MC-DLML vs MCG-DLML (gpu-) running IS.

6.1 Image Segmentation (IS)

Figure 3 shows the performance of MC-DLML and MCG-DLML (gpu-) running IS with 165 image cuts/files, using 2-8 application threads, one of which is the HG thread in MCG-DLML runs. IS is a static application: all the images to be processed are known at the start of each run (no new images are generated dynamically throughout each run). The master application thread inserts the id of each image into its list at the start of computation; then other application threads steal work from the master thread or other application threads that have stolen work.

The figure clearly shows MCG-DLML having much better performance than MC-DLML. This is because IS is an application that is both coarse-grain and compute-intensive. IS is a coarse-grain application in that processing each image is relatively expensive compared to getting its id from a list and reading the image file onto main memory to be processed; IS is compute intensive in that it involves relatively many floating point operations. The use of a GPU by MCG-DLML clearly helps in this kind of workload, since the GPU processes each image in parallel using many GPU threads (one for each pixel), while a core (in the host platform) processes each image sequentially, i.e., all the floating point operations involved in processing an image are carried out sequentially by each core.

The GPU processes most of the workload because it is much faster for this kind of workload than each core: the performance plot of MCG-DLML shows that using only 2 threads (one of which is the HG thread that only schedules work to the GPU) yields as good performance as using more threads. Using 3 threads shows only a marginal performance gain over using 2 threads; using more than 3 threads increasingly deteriorates performance due to synchronisation overhead, particularly the bottleneck of reading image files into main memory.

The performance plot of MC-DLML is typical in that it shows improved performance as more cores (processing elements of same capacity) are used. The best performance corresponds to using the same number of application threads as the number of cores available, 4.

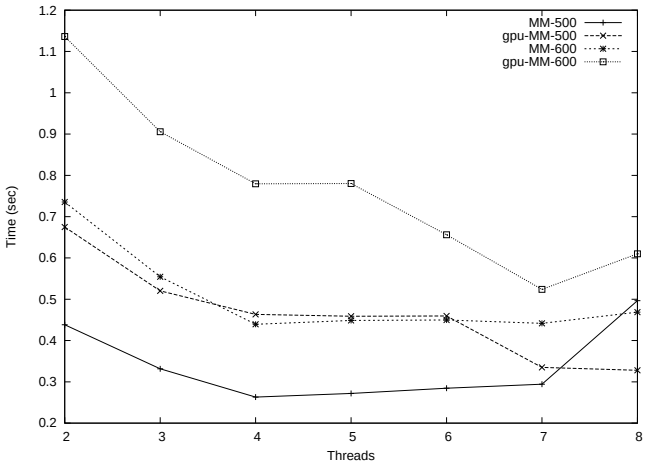


Fig. 4: MC-DLML and MCG-DLML (gpu-) running MM.

6.2 Matrix Multiplication (MM)

Figure 4 shows the performance of MC-DLML and MCG-DLML (gpu-) running MM with $N = 500$ and 600, using 2-8 application threads, one of which is the HG thread in MCG-DLML runs. MM is a static, medium-grain application. MM runs for a relatively short time (less than 1s for those values of N), and thus any of its threads that gets delayed significantly affects overall performance.

MC-DLML performs better than MCG-DLML, nearly twice as fast in most cases. The way we chose to use the GPU within MCG-DLML was not profitable for MM. Recall that each list item contains all the data to compute an element in the results matrix C, i.e.: a full row of matrix A, a full column of matrix B, and the x, y position of the element in C to compute. We decided to move a number of items to the GPU and process each item using only a GPU thread. This is conceptually simple, but it involves a wait to get that number of items to move to the GPU, and also that each GPU thread carries out sequentially all the operations to compute the corresponding element in matrix C.

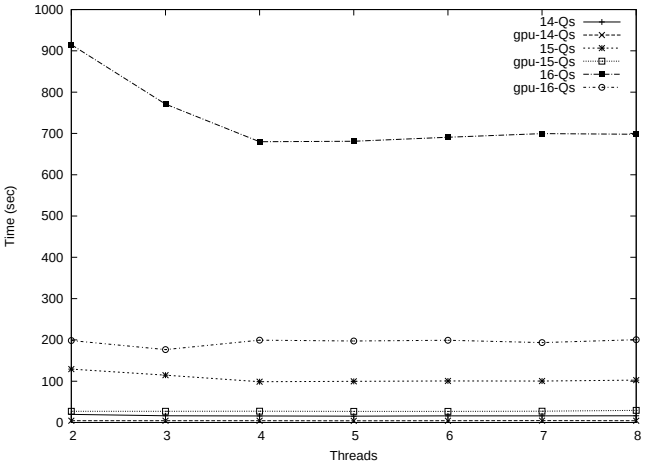


Fig. 5: MC-DLML and MCG-DLML (gpu-) running NAQ.

Such use involves a relatively high communication to computation ratio; the communication consists of that wait to get all the items to move to the GPU at once and the actual copying of the items from the host multicore memory to the GPU memory.

The number of items we are moving from host memory to GPU memory is 5000. The rationale behind this number was to use as many GPU threads as possible and thus make the most out of the GPU without modifying the multicore code substantially. However, this number apparently imposes a coarse granularity in the use of the GPU. We need to investigate further how to better use GPU threads for this kind of workload in the context of MC-DLML.

6.3 The Non-attacking Queens (NAQ) problem

Figure 5 shows the performance of MC-DLML and MCG-DLML running NAQ for $14 - 16$ queens, using 2-8 application threads, one of which is the HG thread in MCG-DLML runs. NAQ is a fine-grain application: processing an item (usually adding a queen) is very quick compared to getting or inserting an item. It is thus prone to high contention overhead. However, NAQ *dynamically* generates new items which each thread inserts into its own list and thus keeps busy doing work.

The figure shows that MCG-DLML significantly outperforms MC-DLML. Despite NAQ being a data-intensive symbolic application, the use of the GPU is very profitable. This is because the communication to computation ratio between (the host multicore and the GPU) is relatively low: the NAQ application generates data dynamically, and most of the data generated by the GPU is processed by the GPU itself. Note that MCG-DLML performs much better MC-DLML for all the problem sizes of the application shown in Figure 5; Figure 6 shows the same results (response times) in logarithmic scale.

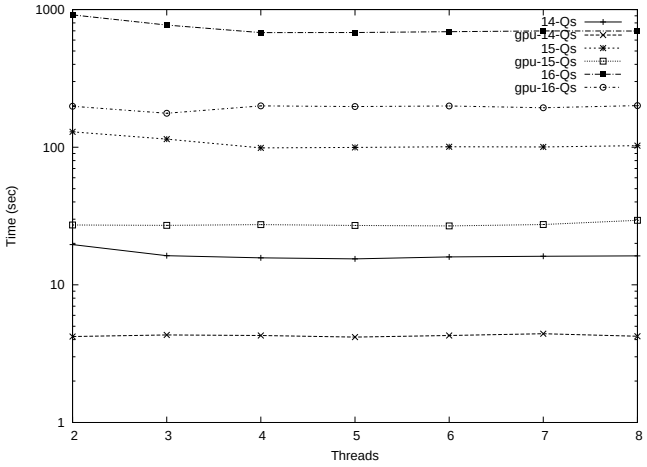


Fig. 6: MC-DLML and MCG-DLML (gpu-) running NAQ, logarithmic scale.

7. Conclusions

We have presented a design and an evaluation of parallel data list processing in multicore-GPU platforms. These platforms offer a very good price-to-computation ratio, and overall our work shows that they can be used profitably both in compute intensive applications (e.g., our IS application) and in symbolic intensive applications (e.g., NAQs application). However, performance may deteriorate substantially because of the communication cost of moving data between the host platform and the GPU device (MM application). Although this is to be expected (in retrospect), we initially thought our MM application would do better running on a mixed multicore-GPU platform (MCG-DLML) than on a single multicore platform (MC-DLML).

The rationale behind the design of (MC-MCG-) DLML is that an application workload gets balanced according to the processing capacity of the processors running an application. It obviously needs further tuning for a multicore-GPU platform. The way a GPU is used imposes a granularity for scheduling work into the GPU, and it can adversely affect performance. We will investigate further how to efficiently use GPUs within MCG-DLML.

References

[1] N. Shavit, "Data structures in the multicore age," *Commun. ACM*, vol. 54, no. 3, pp. 76–84, 2011.

[2] M. Doerksen, S. Solomon, and P. Thulasiraman, "Designing apu oriented scientific computing applications in opencl," in *Proceedings of HPCC-2011: Int'l Conf. on High Performance Computing and Communications*, IEEE, Ed., 2011, pp. 587–592.

[3] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, vol. IN PRESS.

[4] J. Buenabad-Chávez, M. A. Castro-García, J. L. Quiroz-Fabián, D. M. Yellin, G. Román-Alonso, and E. F. Hernández-Ventura, "Low-synchronisation work stealing under parallel data-list processing in multicores," in *Proceedings of PDPTA-2011: 17th Int'l Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, 18-21 July 2011, pp. 850–856.

[5] J. Buenabad-Chávez, M. A. Castro-García, J. L. Quiroz-Fabián, E. F. Hernández-Ventura, G. Román-Alonso, D. M. Yellin, and M. Aguilar-Cornejo, "Reducing communication overhead under parallel list processing in multicore clusters," in *Proceedings of CCE-2011: 8th Int'l Conference on Electrical Engineering, Computing Science and Automatic Control*, Merida Yucatan, Mexico, 26-28 October 2011, pp. 780–785.

[6] A. Bruen and R. Dixon, "The n-queens problem," *Discrete Mathematics*, vol. 12, pp. 393–395, 1975.

[7] M. A. Castro-García, "Programación con listas de datos para cómputo paralelo en clusters," Ph.D. dissertation, CINVESTAV-IPN, México 07360, D.F., 2007.

[8] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.

[9] J. Dean and S. Ghemawat, "Mapreduce: Simplifed data processing on large clusters," in *Operating Systems Design and Implementation*, 2004, pp. 137–149.

[10] J. Buenabad-Chávez, M. A. Castro-García, and G. Román-Alonso, "Simple, list-based parallel programming with transparent load balancing." in *Parallel Processing and Applied Mathematics*, ser. LNCS, vol. 3911. Springer, 2005, pp. 920–927.

[11] NVIDIA, *CUDA Programming Guide Version 2.3.1*. NVIDIA, 2009.

[12] J. R. Jiménez-Alaniz, V. Medina-Bañuelos, and O. Yáñez-Suárez, "Data-driven brain mri segmentation supported on edge confidence and a priori tissue information," *IEEE Trans. on Medical Imaging*, vol. 25, no. 1, pp. 74–83, January 2006.

[13] G. Román-Alonso, J. R. Jiménez-Alaniz, J. Buenabad-Chávez, M. A. Castro-García, and A. H. Vargas-Rodríguez, "Segmentation of brain image volumes using the data list management library," in *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*, 2007, pp. 2085–2088.

Tasks Merging Technique for Optimization of Scheduling

Marjan Abdeyazdan

Department of Computer Engineering, Mahshahr branch, Islamic Azad Universiy

Mahshahr, Iran

Phone : 0098-916-152-0433

E-mail: abdeyazdan87@yahoo.com

ABSTRACT

The issue of static scheduling of tasks is of particular importance in multiprocessor systems due to optimal use of processors and taking less time. It is a NP-Hard problem and obtaining the optimal solution involves a high time complexity. Thus the heuristic method is used to solve this problem. The genetic algorithms and Automata are suitable method for scheduling of multiprocessor systems. In this paper a solution is introduced for algorithms of scheduling. The main idea in designing these algorithms is to obtain the least run time and the highest possible parallel with minimum critical path. By using of this solution the graphs with many tasks can be transformed in to fewer ones. We use a combination of both tasks merging technique and scheduling with genetic of algorithm to shorten the length of critical path.

A new genetic algorithm is introduced to schedule of multiprocessor systems in which the scheduling priority of performing the tasks is based on the number of child. The results show that the new proposed algorithm achieves the optimal scheduling solution in acceptable time compared to other common methods.

Keywords

Scheduling, tasks gragh, merging, Multiprocessors, genetic algorithm.

1. INTRODUCTION

The issue of scheduling in multiprocessor systems and heterogeneity distributed systems is a NP –Hard problem. In the classical method it is time-consuming to obtain of a complete optimal solution of a minimum scheduling and in many cases running the tasks randomly takes too much time. Thus the heuristic complete optimal solution is not necessarily obtained but within reasonable time span an answer will be obtained which almost close to optimal solution. Many heuristic methods have been introduced include: Duplex[14,12], OLB[14,12,4], GA[19,20,17,24,21], SA[1,2,19,13,3], LMT[11], MCT[14], A*[16,2,19,15], Min-min[14,12,10], Max-min[14,12,10], DLS[5], GSA[13], Tabu search[18,19]. One of the best heuristic method is to use the genetic algorithm. In this paper a new genetic algorithm based on earliest tasks run is simulated given their priority on the number of child.

In order to increase the program efficiencies in parallel systems an accurate scheduling of tasks is highly taken into consideration. The scheduling should be performed so that the total run time of the program with regard to tasks run time and the relation between processors can be minimized. Certainly the more the number of processors during running, the simpler the planning carries out. Today it is common to use a lot of processors in super computers, quick processing clusters, the distributed networks like grids. Hence it is important to optimize along the critical path. By using the tasks merging the length of this path can be decreased considerably.

Given being NP-Complete of the scheduling task the approaches based on uncertainty methods in this field will not ever efficient. In the past decades much works have been performed by this algorithm, however many of which had not the required efficiency. Therefore, use of uncertainly method and mainly the genetic algorithm is very effective for solving such problems.

In this paper, we use a combination of both task merging technique and scheduling in order to shorten the critical path. Accurately merging of the tasks before scheduling the task graphs have significant impact on scheduling result. The results obtained by applying our algorithm on the different graphs for evaluating the scheduling algorithm represents a positive impact of merging on scheduling.

2. RELATED WORKS

In order to improve the performance of scheduling algorithms of task graphs, before applying the algorithms, they can be prepared by using two solutions i.e. clustering [6]and tasks merging [22,26] for suitable scheduling. The results from multi processors scheduling algorithms can be greatly improved by using of both solutions. In the following, the two above solutions will be examined to solve multiprocessors scheduling problems.

A lot of works have been done on task scheduling in the multiprocessor systems by genetic algorithm [19,20,17],. Some of them have employed the random genetic algorithm and some others used genetic algorithm and prioritizing tasks based on height [20]. In this research a new genetic algorithm has been simulated based on earliest task run given their priority and also on the number of their child.

2.1. Clustering the tasks

A task clustering solves only a part of multiprocessor scheduling problem. A cluster of tasks is a set of tasks that together are run on a processor. In turn all the edges inside a cluster of a graph their relation cost equals to zero. Although the sequence of running tasks within a cluster is often determined by algorithms of clustering, the main aim of them is to increase granulating of tasks graph. However the tasks within clusters are not summed together. In the other word, the resulting data from each of tasks are sent as soon as provided. It is considered as a substantial difference between task clustering technique and task merging [2].

In order to cluster the tasks some optimal or nearly optimal processors are determine for task graph scheduler. In the other word, the more processors are needed from generated clusters by algorithm of task scheduling. Also it shows where the task clustering is performed prior to scheduling. The balancing (LB) for merged cluster is proffered because LB is fast and is easily implemented and yields better final scheduling. In short, two phase method of jobs scheduling and balancing have significant effect strategically for job graph scheduling on distributed parallel architecture.

2.2. Algorithm of Merging Task

In [4] a merging task algorithm is presented which its input is a graph with micro granulating. The main idea is iterative selection of a pair node for merging. If merging of these two nodes lead to decrease the length of critical path, then it select two new nodes.

2.3. Algorithm Based on Graph Rewriting System

Graph Rewriting System (GRS) Is an algorithm used for merging tasks[22]. It is employed as a way for reducing the length of critical path. The transformation is specified by several graph rewriting rules.

2.4. Prioritizing scheduling algorithm based on height or based on offspring (children)

The aim of scheduling in a multiprocessors system is to assign N job to M processor so that the finished run time of final job in this system is minimized. For simplicity, if two tasks are scheduled on two different processors, then the communication cost for sending data between two tasks is taken as zero. The algorithm of task scheduling generation in term of height or offspring is as follows:

a. Arrange the tasks in order of incremental in term of their height or offspring (children).

b. Repeat the stages 3,4 in the tasks.

c. Generate a random number r, so that 1<r<m (m is the number of processors)

d. Select the first task from arranged ones and allocate it to processor r^{th} then eliminate it.

By repeating the algorithm of scheduling generated, an initial population is generated from search nodes.The path from input node to output node is the longest critical path where we denote it by and length of path by representing the run time of the program in a state in which the maximum number of processors can be used. (Each existing task in task graph is sent to a processor) the value of upper bound of output node reflects the length of critical path of the graph.

3. PROPOSED ALGORITHM

This novel scheduling algorithm is a combined technique resulting from task merging as well as tasks scheduling. In order to manage large graph with micro granulating, the tasks are merged so that in addition to improve the graph specifications such as length of critical path, the amount of granulating is increased as well. It is done through graph rewriting system which consist the series of codes transforming graph on task graph.

It is NP-Complete. Therefore, the approaches based on conclusive methods are not so efficient in this field. Using of evolutionary processing algorithm and mainly genetic algorithm given their non-conclusive nature will be effective for solving problems. Genetic scheduling algorithm is ever seeking to find the closest answer. In order to find a closest answer, this paper presents a two-step algorithm as follows:

The first step is to receive a task graph with micro granulating. After receiving the task merging algorithm graph, it analyzes the graph and attempts to merge the tasks for modifying the parameters of qualitative program. The next one, the obtained graph from this merging is carried out by priority genetic scheduling algorithm based on child. The cost sending data between the tasks equals to L+n/B where n is the delivered data cost, (B) is the bandwidth an (L) is the delay time of the dispatched data. Similarity the scheduling problem can be considered as triple (G,B,L) in which G is the task graph, B is bandwidth and L is the delay time.

In the general state, the algorithm of rewriting graph system can be taken into account as three independent rules: 1: Merging single children, 2: Merging all parents, 3: Replicate Parent Merge .

The rule of Merging single children has the lowest cost since it does not involve any condition but a sub-graph pattern (figure 1).

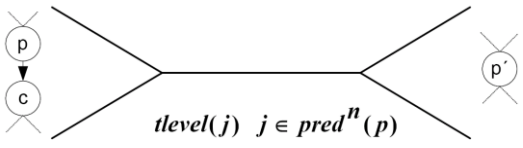


Figure 1. Merging single children

As shown in figure 1, by using this rule we can merge two nodes with each other if in the tasks graph a node has a single children and its children owing a single parent. In such conditions, the node C for running will wait the results

from node p. If this merging is failed these two nodes may be assigned to two distinct processors in a scheduling and it is obvious that a such action only will lead to delayed entering in to system as large as edge weight connecting between both nodes C and p. By merging two above mentioned nodes the possibility of wrong decisions will be removed when scheduling tasks graph.

At first glance, it seems that if using of an ideal scheduling algorithm taking such implications in to account it can be replaced by above method. However by now such algorithm is not introduced because of high complexity of scheduling. In this paper by distinguishing the problem of tasks graph quality improvement from scheduling one the complexity of the problem has reduced and the possibility of actual implementation has also provided.

The merging of propagated parent includes second merging all parent rule (figure2). In this rule a task of parent is divided in to a number of tasks based on the number of its child and each of these copies is merged by one of the child. The first row in the list of conditions will be decisive so that the upper bound for each children and their child is not increased by merging copy p.

In the other word if the greatest delay factor of running the nodes c_i is the node p, then by merging it this amount can be reduced (set of nodes C) for each of child having such situation.

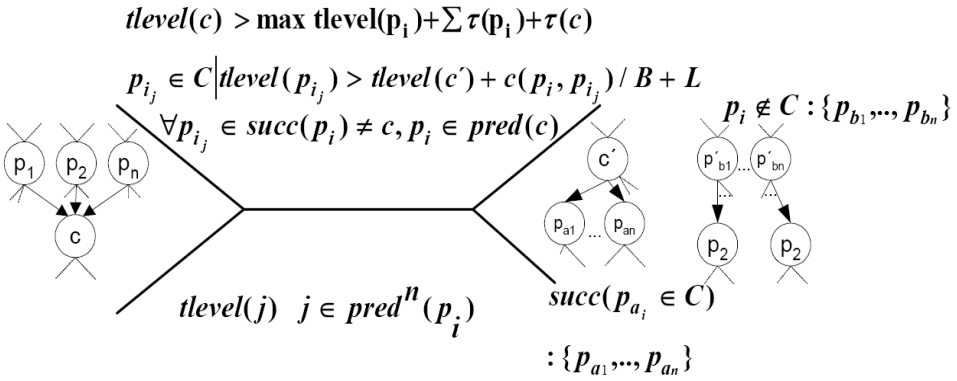


Figure 2. Merging all parents

As shown in figure 2, by using this rule we can If the node p is not the factor affecting the maximum delay of running child c_i, though by merging the node p with nodes c_i the upper bound of that node is not increased, but due to increased volume of running the negative impact of upper bound is transferred to child c_i. Thus, as shown in figure 2, the node p is propagated for the whole child existing within C set and merged in to them; if the node c_i does not exist in C set it will be left alone.

The last rule is the merging of the whole parents As shown in figure 3. Here all of the parents' child of C node denoted by p_{ij} are examined. If merging the whole parents with C causes the level of upper bound of these child is not increased the node of interest is added to C set.

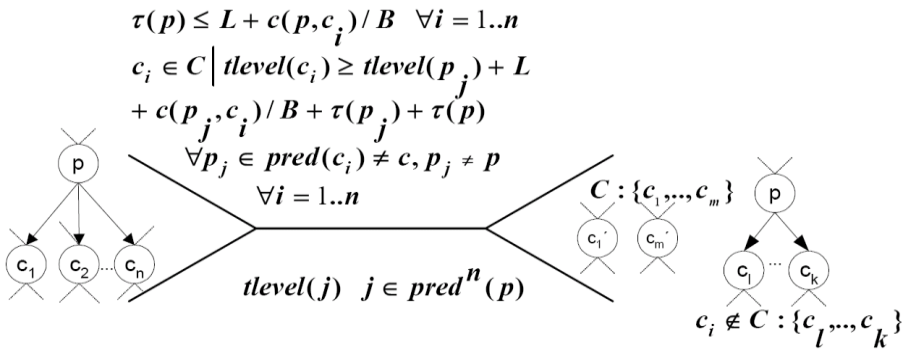


Figure 3. Replicate Parent Merge

The algorithm generating tasks scheduling for initial population in term of the number of offspring is as follows[25]:

a. Arrange the tasks in the order of descending in term of the number of their offspring.

b. Put separately the task with the same number of offspring.

c. Select a task from group of tasks randomly and eliminate it from group.

By using of this algorithm the degree of task graph parallelization will be increased due to decrease the length of critical path. It means that we can obtain a more suitable solution by increasing the number of processors. Having applied the above rules, the graph is prepared for applying the scheduling. The graph resulting from this technique is introduced as an input to prioritized genetic scheduling algorithm based on the number of offspring for scheduling. In the algorithm evaluation section it has shown that by using of this method a suitable scheduling will be established. The proposed method passes the tasks based on prioritizing the new offspring on processors in order to be run. It means that the more the tasks have offspring the sooner scheduling is done. Then that tasks are dedicated to processors P1 to Pm based on EST method in such a way that start time for that task on the processors became short for other processors.

4. SIMULATION AND EVALUATION

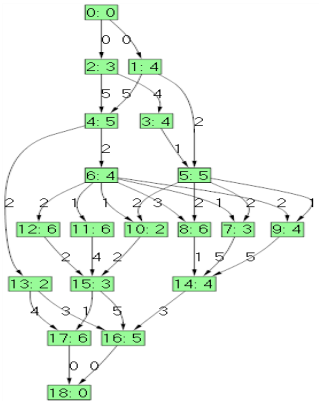


Figure 4. A standard task graph

Table 1. Comparison of the figure 4 graph with existing scheduling algorithm.

MCP	LAST	ETF	HLFET	EZ	LC	NOVIN
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

In order to evaluate the proposed algorithm, a series of simulation has been performed by Visual Basic.net 2005 package on a computer Pentium IV equipped with processor AMD 2.8 MHZ with 512 megabyte memory. By using a formulated program for automatically generating graph, a set of task graphs including 57 graphs with the number of 30, 70, 90 tasks with tasks dependency percentage between 1 to 100 have been established in such a way that the task run time can be varied in the range of 1 to 1000. These graphs were scheduled on multiprocessors with the number 5,7,9 ones for genetic algorithm without applying any priority, genetic algorithm based on offspring

In order to evaluate the solution presented in this paper the priority genetic scheduling algorithm based on offspring is used and also the graph resulting from merging is assessed. The scheduling problem is NP-Hard and so far many algorithms have been suggested for it [27,6]. The above optimizing algorithm is applicable before any scheduling, and can be used in place of proposed genetic algorithm.

The solution of the problem or completed time of running the final task in the scheduling algorithm after averaging for any similar number of processors / number of work is shown in table 2. By over viewing the results it is seen that the proposed algorithm has a solution close to optimal one.

By running the proposed algorithms on graph in the figure4, a graph with 16% modification in parallel as well as 6% improvement in length of critical path will be obtained. Also 5% decrease is seen in number of task graph nodes. Figure 5 shows algorithm stability.

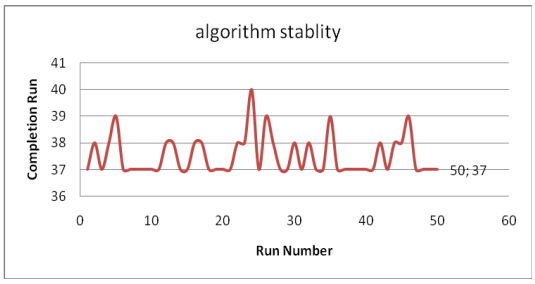


Figure 5. Algorithm stability

This graph is introduced into scheduling algorithm as an input. A run time 38 is obtained with six processors. In the table 1 and figure 6, we see the results of graph figure 4 with algorithms ETF [9], HLFET [7], MCP [8], EZ [16],LC [10].

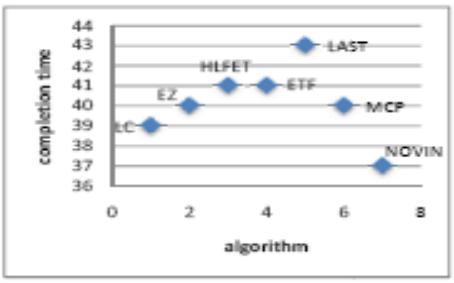


Figure 6. Comparison of the figure 4 graph with existing scheduling algorithm

as well as new proposed algorithms. In order to take other thing being equal, the experiment of all the running steps and operators have been considered the same for all the three algorithms except for prioritizing and establishing initial population. The values of parameters of initial population for algorithms are the number of tasks and replications (number of generations) respectively. Prioritizing based on the number of offspring is better than on the height because it is possible that a task has higher height than the other one and at the same time has more offspring. Logically, running the tasks with more offspring have a greater priority compared to the ones having lower height. It allows more tasks to be run, because its prior task has been done. On the other hand, it is impossible, that the task has more offspring to be the task with less offspring. Therefore, its running does not depend on finishing the task with less offspring.

The solution of the problem or completed time of running the final task in the scheduling algorithm after averaging for any similar number of processors / number of tasks is shown in table 2. By over viewing the results it is seen that the proposed algorithm improves the solution close to optimal one 39% and 24 % compared to priority based on height algorithm and 17 % compared to random genetic algorithm and priority genetic algorithm in term of number of offspring respectively. However, the computing time and implementation of the four algorithms are within the same place since only the parts of the four algorithms differ with each other; each is computed once and initially for generation of primary population.

Table 2. The schedules for last algorithms and proposed algorithm (Merge and Priority based on children)

Algorithms	Total Finish Time (TFT) (seconds)								
	Number of processors = 3			Number of processors = 5			Number of processors = 7		
	Number of tasks			Number of tasks			Number of tasks		
	30	50	70	30	50	70	30	50	70
Algorithm Genetic(A)	1066	2729	7223	2604	2135	3498	855	2314	2919
Priority based on height(B)	978	2440	6003	1882	2005	3222	850	2199	2888
Priority based on children(C)	918	2300	5792	2222	1981	3197	809	2065	2742
Merge and Priority based on children(D)	803	1947	4475	1491	1644	2588	736	1789	2251
Optimal percent (D) to (A)	32	40	61	74	30	35	16	29	30
Optimal percent (D) to (B)	21	25	34	26	21	24	15	22	28
Optimal percent (D) to (C)	14	18	29	5	20	23	10	15	21

Table 2 shows the schedules and TFT for four scheduling algorithms. The results indicate that our suggested new algorithm finds better schedule with minimum TFT compared to the other heuristics. While the computation time of the two above genetic algorithms is more than the other heuristics obviously, and is quite similar, as only their initial population producing step is different, the step is calculated once.As it is shown, while the number of tasks, the number of processors and task implementation within specified time are taken constant.

As the task dependency percentage increases, corresponding to proposed algorithm optimized percentage for obtaining the answer close to optimal one, it increases over the two other algorithms. The main reason of this feather is attributed to merge the tasks prior to prioritize the tasks based on the number of offspring. Since the higher task dependency percentage is taken into account, the more merging of tasks and offspring is and the scheduling became better.

5. CONCLUSION

In this paper, a method is presented for improvement of scheduling algorithm efficiency on graph through technique of merging tasks GRS. First the existing rules in GRS are applied on graph of interest then the output is given to genetic scheduling algorithm based on the number offspring (children). The results from applying the proposed solution on graphs show that after merging the task, the existing tasks within graph become small, the parallel increases and the length of critical path is shorten.

6. ACKNOWLEDGMENT

This work is supported by mahshahr branch islamic azad university and it is a scientific design.

7. REFRENCES

[1]. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, Optimization by simulated annealing, Science 220, 4598 (May 1983), 671_680.

[2]. S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Prentice_ Hall, Englewood Cliffs, NJ, 1995.

[3]. A.Y. Zomaya and R. Kazman, Simulated annealing techniques, in "Algorithms and Theory of Computation Handbook" (M. J. Atallah, Ed.), pp. 37-1_37-19, CRC Press, Boca Raton, FL, 1999.

[4]. R. F. Freund and H. J. Siegel, Heterogeneous processing, IEEE Compute. 26.6 (June 1993), 13_17.

[5]. G.C. Sih and E.A. Lee, A compile- time scheduling heuristic for interconnection- constrained heterogeneous processor architectures, IEEE Trans. Parallel Distrib. Systems 4 (Feb. 1993), 175_186.

[6]. Y. K. Kwok and I. Ahmad ,"Static Scheduling Algorithms for Allocating Directed Task Graph Multiprocessors", ACM Cumputing surveys,Vol. December 1999.

[7]. T.L. Adam, K.M. Chandy, J. Dickson, "A Comoarison of List Scheduling for Parallel Processing Systems," communications of the ACM, vol. 17, DEC. 1974, PP. 685-690.

[8]. M.-Y. WU, D.D. Gajski, "Hypertool: A Programming aid for Message-Passing Systems," IEEE transactions on parallel and distributed systems, vol. 1, NO. 3, JULY 1990, PP.330-343.

[9]. J.J. Hwang, Y.C. Chow, F.D. Anger, C.Y. Lee, "Scheduling Precedence Graphs in Systems with Interproccessor Commonication Times, " siam journal on computing, vol. 18, NO. 2, APR. 1989, PP. 244-257.

[10]. S.J. Kim, J.C. Brown, "A General Approach to Mapping of Prallel Computstion Upon Multiprocessor Architectures, " procedings of international conference on parallel processing, vol. II, AUG. 1988, PP. 1-8.

[11]. Boontee Kruatrachue, Ted Lewis,"Grain Size Determination for Parallel Processing", IEEE 1988.

[12]. R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd. M Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, Scheduling resources in multi- user, heterogeneous, computing environments with SmartNet, in :7th IEEE Heterogeneous Computing Workshop (HCW'98), "pp. 184_199, 1998), 13-17.

[13]. P. Shroff, D. Watson, N. Flann, and R. Freund, Genetic simulated annealing for scheduling data- dependent tasks in heterogeneous environments, in "5th IEEE Heterogeneous Computing Workshop (HCW '96)," pp. 98_104, 1996.

[14]. R. Armstrong, D. Hensgen, and T. Kidd, The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions, in 7th IEEE Heterogeneous Computing Workshop (HCW, 98), "PP. 79-87, 1998.

[15]. C.-C. Shen and W.-H. Tsai, A graph matching approach to optimal task assignment in distributed computing system using a minimax criterion, IEEE Trans. Comput. 34, 3 (Mar. 1985), 197_203.

[16]. V. Sarkar, ``Partitioning and Scheduling Parallel Programs for Multiprocessors," MIT press, cambridge, MA, 1989.

[17]. H. Singh and A. Youssef, Mapping and scheduling heterogeneous task graphs using genetic algorithms, in "5th IEEE Heterogeneous Computing Workshop (HCW '96)," pp. 86_97, 1996.

[18]. F. Glover and M. Laguna, Tabu Search, Kluwer Academic, Boston, MA, 1997.

[19]. B. Naharai, A. Youssef, and H. A. Choi, Matching and scheduling in a generalized optimal selection theory, in "Proceedings of the Heterogeneous Computing Workshop," pp. 3-8, April 1994.

[20]. Lee, Y. H., Chen, C. A Modified Genetic Algorithm for Task Scheduling in Multi processor Systems, the ninth workshop on compiler techniques for high performance computing, 2003.

[21]. m. h. shenassa, m. mahmoodi, a novel intelligent method for task scheduling in multiprocessor systems using genetic algorithm, journal of Franklin institute, Elsevier, 2006.

[22]. Peter Aronosson, Peter Fritzson," Task Merging and Reolization Using Graph Rewriting' 2002 .

[23]. M. Ayed, J-L Gaudiot," An Efficint Heuristic for Code Partitioning', parallel computing, 26:399–426, 2000.

[24]. m. h. Shenassa, m. mahmoodi, a new intelligent method for task scheduling in multiprocessor systems using genetic algorithm, Proceeding of the First International Conference on Modeling, Simulation and Applied Optimization, Sharjah, U.A.E. February 1-3, 2005.

[25]. Marjan Abdeyazdan, Amir Masoud Rahmani, " Multiprocessor Task Scheduling using a new Prioritizing Genetic Algorithm based on number of Task Children ", 7th International Conference on Distributed and Parallel systems DAPSYS' 2008 Debrecen, Hungary, September 3 – 5, 2008.

[26]. Peter Aronosson, Peter Fritzson," A Task Merging Technique for Paralllization of Modelica Models", march 2005.

[27]. Marjan Abdeyazdan, Amir Masoud Rahmani, Vahid Arjman, Hamid Raeis Ghanavati, "Genetic Algorithm based on number of children and height task for multiprocessor task Scheduling" WORLD COMP" 11 Springer 2011.

SESSION

SYSTEMS SOFTWARE + OS + THREADS + CACHING + PROGRAMMING MODELS AND LANGUAGES + I/O and ARCHITECTURE ISSUES

Chair(s)

TBA

Locality-aware memory system for PRAM mode private data storage in the CESM architecture

Martti Forsell
Platform Architectures Team
VTT
Oulu, Finland

Abstract - *The Configurable Emulated Shared Memory (CESM) is a chip multiprocessor (CMP) architecture. It implements a dual computational model—the strong synchronous Parallel Random Access Machine (PRAM) and weaker but more conventional asynchronous Non-Uniform Memory Access (NUMA)—so that a programmer can make use of easy-to-use fine-grained parallel algorithmics of the PRAM paradigm for functionalities with enough parallelism and standard coarse-grained NUMA algorithms for low parallelism cases. In this paper we propose adding local memories to CESM processors for storing thread-private data in the PRAM mode for performance and safety reasons. While this kind of solutions typically introduce a partitioning problem, most parallel computing languages make use of the concepts of shared and private data that provide natural and orthogonal partitioning of data to global and local subspaces. Preliminary performance evaluation of the proposed solution on our CESM architecture along with programming/implementation considerations are provided.*

Keywords: Parallel computing, CESM, PRAM, memory system, local memories, private and shared data

1 Introduction

Although all the processor manufacturers have moved to multicore processors, programming of them has remained challenging. This is partially because the computational models defined by the underlying architectures are asynchronous and locality-sensitive, forcing a programmer to constantly orchestrate execution threads with barrier synchronizations and locks and play with functionality mapping schemes and data partitioning to obtain even decent performance. The rest of the challenge is due to the fact that by far the most programming-related education is given using the sequential computing paradigm. Instead of trying to incrementally refine the current tools and the architectures, we take a radical approach of studying new architectures that would provide stronger models of computation and make it possible to program multicore machines in a much simpler way.

The *Configurable Emulated Shared Memory* (CESM) is a *chip multiprocessor* (CMP) architecture [Forsell09]. It implements a dual computational model—the strong synchronous *Parallel Random Access Machine* (PRAM) [Fortune78] and weaker but more conventional asynchronous *Non-Uniform Memory Access* (NUMA) [Swan77] scheme—so that a programmer can make use of easy-to-use fine-grained parallel algorithmics of the PRAM paradigm for functionalities with enough parallelism and standard coarse-grained NUMA algorithms for low parallelism cases.

In this paper we propose adding/making use of local memories to CESM processors for storing thread-private data in the PRAM mode for performance and safety reasons. While this kind of solutions would typically introduce a partitioning problem, most parallel computing languages make use of the concepts of shared and private data that provide natural orthogonal partitioning of data to global and local subspaces. Preliminary performance evaluation of the proposed solution on our CESM architecture along with programming/implementation considerations are provided.

The rest of the article is organized so that in Section 2 we describe the basic concepts used in this paper, including the PRAM-NUMA and CESM, the way how local memories are employed in the PRAM mode is described in Section3, in Section 4 we discuss language and compiling support with practical examples, in Section 5 we evaluate the performance of the proposed solution on our CESM architecture, and finally in Section 6 we give our conclusions.

2. Basic concepts

In order to extend the success story of the sequential computing paradigm also to parallel computing, we have developed the CESM architecture. It realizes the PRAM-NUMA model of computation, which can be seen as a physically feasible extension of the model of sequential computation. In the following the basic concepts related to this attempt are explained.

2.1 PRAM-NUMA

The PRAM-NUMA model of computation is a configurable synchronous shared memory model developed by us [Forsell11a]. It consists of T processors grouped as P groups of T_p processors, a word-wise accessible global shared memory, P local memory blocks, a *metric* defining distance between the processor groups and target memory blocks, and distance-aware interconnection network (see Figure 1). Each processor is attached to the shared memory and each processor group is attached to its own local memory block. The interconnection network connects the local memory access paths of processor

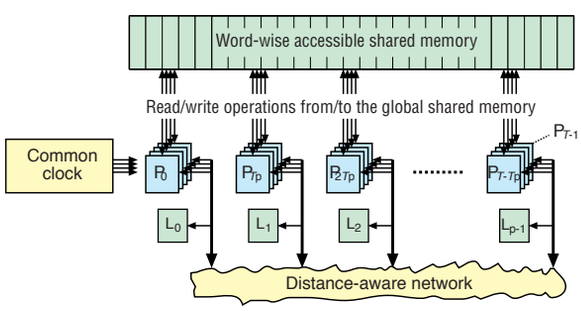


Figure 1. PRAM-NUMA model (P=processor, L=local memory).

groups together so that remote local memories can be accessed. This network is distance-aware in a sense that the latency of routing is proportional to the distance between the source processor and destination memory. The bandwidths from a group of processors to the shared memory and local memory are the same. Each processor can be *configured* to either the PRAM mode or NUMA mode. Along with configuration from the PRAM mode to the NUMA mode, one can set the state of the processor to point an arbitrary state within the group it belongs to. With this indirection of states, two or more processors belonging to the same group can be configured to form a NUMA bunch so that they execute a common instruction stream and share their state with each other, i.e. can be seen as executing code like a single processor would do.

Execution in the PRAM-NUMA model happens in synchronous *steps* during which every processor of each group executes exactly one instruction in the PRAM mode and every bunch executes exactly as many instructions for a single instruction stream as there are participating threads in the bunch.

The PRAM-NUMA model solves the low-TLP execution problem of standard *Emulated Shared Memory* (ESM) machines by providing a possibility to configure two or more processors of a group to use a common state like they were a single processor and to perform NUMA access to the local memories. In the case of low TLP portion of code, a programmer can just set up to T_p processors per group to run that portion as a NUMA bunch and gain more performance proportionally to the number of processors in the bunch.

2.2 CESM

The CESM architecture is a hybrid architecture implementing the PRAM-NUMA model of computation to support both high-speed execution of functionalities with sufficient parallelism and full utilization in low parallelism cases [Forsell09]. A CESM CMP consists of P T_p-threaded (in total constituting $T = PT_p$ threads) F-functional unit *MultiBunched/Threaded Architecture with Chaining* (MBTAC) processor cores [Forsell10] connected to a distributed memory system (see Figure 2). The memory system has P dedicated instruction memory modules, P local data memory modules, P T_p-line step caches and scratchpads attached to processors, P fast data memory modules with active memory units, and a high-bandwidth multimesh interconnection network. Step caches, scratchpads and active memory units are used to support concurrent memory access and multioperations in the PRAM mode [Forsell06]. A MBTAC processor features A ALUs, M memory units, compare unit, and sequencer organized as a chain for the PRAM mode and a single ALU, memory unit, and sequencer organized in parallel for the NUMA mode. In order to save in hardware costs and to provide as seamless configurability as possible between the PRAM and NUMA models, the execution pipelines for the modes are merged and share some units like fetcher, operand select, and the first ALU (see Figure 3). The effective length of the pipeline is T_p for the PRAM mode and 4 for the NUMA mode. The CESM architecture implements support for fast and synchronous switching between the PRAM and NUMA modes for groups of threads with dedicated machine language instructions JOIN and SPLIT [Forsell09]. Synchronous switching is necessary because a NUMA bunch may get switched back to the PRAM mode in the middle of a PRAM step.

3. Supporting PRAM mode local access

In order to increase the performance of processors, providing sufficient memory system bandwidth is often one of the most critical things. In the following we propose adding local memory access to the PRAM mode of the CESM architecture.

3.1 PRAM mode local access

At the first glance, adding local memories to a PRAM-based architecture may sound to be against the basic idea of the model. If we however take a more detailed look at the models at hand, we can observe that the PRAM-NUMA model features already local memories but they are dedicated for local/distance aware remote access in the NUMA mode only. Since also our PRAM-NUMA implementation architecture, CESM, implements local memories for the NUMA mode and connects them together with a help of the PRAM mode intercommunication network, it is quite straightforward to consider using these local memories also in the PRAM mode. This does not require extra hardware except for adding the local memory operation code field to the PRAM mode instruction word and adding logic that controls the local memory unit with this code in the PRAM mode (see Figure 3).

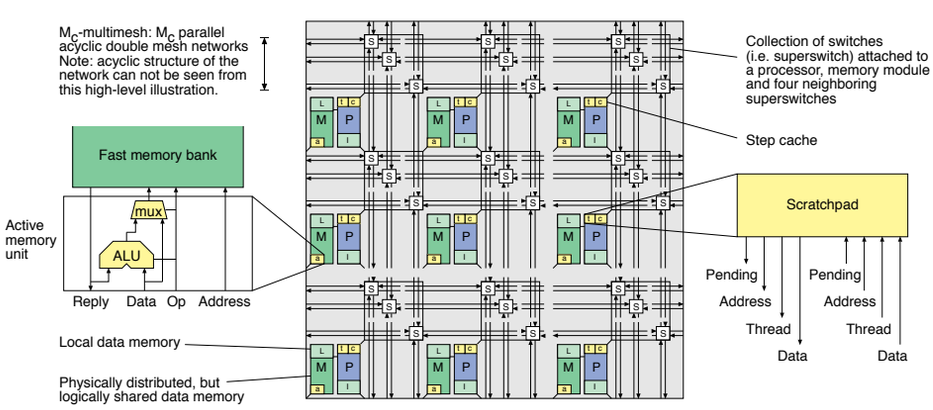


Figure 2. Block diagram of the CESM architecture (P=processor, M=shared data memory, L=local data memory, I=instruction memory, a=active memory unit, c=step cache, and t=scratchpad)..

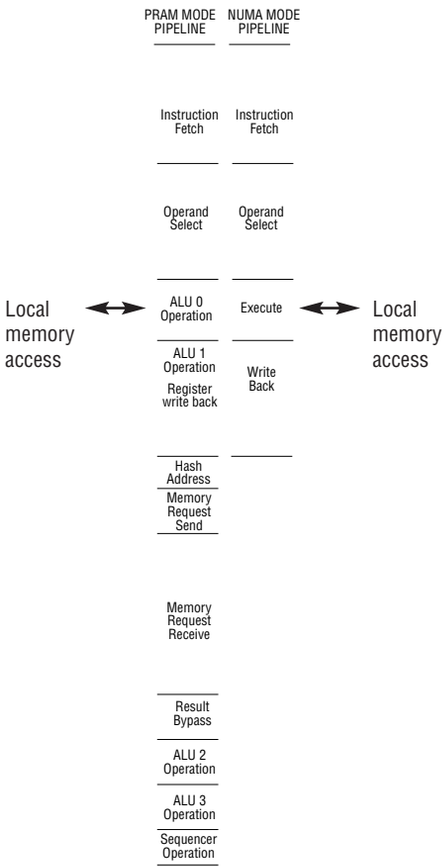


Figure 3. Pipeline for the local memory aware solution.

The local memory space of the CESM is easier to extend with techniques used in sequential computers than the shared memory. One can just provide each local memory with a standard memory hierarchy with multiple levels of caches like in the sequential machines. There will be no coherency problems since the local memories are distinct from each others assuming the standard NUMA model is used.

In theory it is possible to extend this technique to multiple local memory units per processor core but this would make the partitioning problem much worse.

3.2 Related work

To our best knowledge, this is the first attempt to add local memories to an ESM architecture. However, there exist some prior work for the ESM architectures as well as for the memory systems for them.

The shared memory emulation was first attempted in the late 70's soon after the PRAM model was introduced [Schwarz80, Gajski83, Pfister85]. These attempts, however left much room for improvements in performance, implementability and scalability. The second wave of attempts [Alverson90, Abolhassan03] included much more sophisticated techniques to address synchronicity and bandwidth constraints of existing parallel architectures. The current ESM projects include the TOTAL ECLIPSE [Forsell02a, Forsell10] and XMT [Vishkin11]. The former provides fully scalable sparse/multimesh interconnection network while the latter relies on low-latency mesh of trees network featuring link length proportional to square root of P divided by logarithm of P and thus having weaker scalability. They both aim to implement ESM as a *Chip MultiProcessor* (CMP) making use of the huge communication potential of on-chip networks.

The memory systems for ESM computers has been studied in the TOTAL ECLIPSE research line. Balancing the slow data memory access/cycle time with the processor clock rate is considered in [Forsell02b]. Providing sufficient bandwidth to the instruction memory in ESM is studied in [Forsell02c]. Active memory techniques supporting the partial and full multioperation support are considered in [Forsell05] and [Forsell06], respectively. Local memories are used in many commercial application-specific processors, e.g. DSPs, to improve the memory bandwidth.

Intercommunication belonging to essential parts of memory systems of ESM computers is considered in a wide variety of studies. For a summary of the work done before the year 1996, [Leppänen96] reviews communication solutions that can be used for PRAM realization. Our recent study [Forsell11b] evaluates the bandwidth, silicon area, and power consumption of various sparse networks, e.g. sparse meshes, making use of multidimensional embeddings. Various aspects of memory systems in ESMs making used of networks on chip are studied in [Forsell11c].

4. Methodological considerations

Widely used distinction between shared and thread-private data is the key mechanism for simple exploitation of the local memories in the PRAM mode. In the following we will take a look at needed language and compiler support.

4.1 Language support

Most parallel programming languages make use of the concept of sharity of a variable. The two most popular forms of sharity are shared and private. A *shared* variable is accessible by all the execution threads while a *private* variable can be accesses only by a single thread for which it is declared. The former raises the question of concurrent access models in synchronous architectures and exact timing and order of accesses in asynchronous architecture. In the synchronous PRAM mode of a CESM architecture a programming language could make use of this distinction to provide a natural partitioning of data to global and local memories. With these assumptions, programming the proposed CESM architecture with a high-level parallel language happens exactly in the same way as that of the baseline CESM machines. Explicit assembler programming is a bit more difficult than that of the baseline machine since one must assign data to corresponding global and local memory units explicitly.

Consider functionality computing the *prefix sum* of a vector of integers. With the high-level e-language, it can be easily implemented for the PRAM mode of CESM (see Figure 4a). The compiled version with explicit memory operation assignments is shown in Figures 4b and 4c. Since global and local memory accesses can be made during a single step for each thread, the number of the instructions in the inner loop (L5_LF0) is reduced by 20%. The speedup is not higher in this case since the local variable data is stored in a register making the data storage always private.

(A) The E-LANGUAGE source

```
#include "e+.h"
#define   size 65536
int sum_=0;                                          // Shared variable
int source_[size];                                   // Shared array
int main()
{     int i,p;                                       // Private variables
      for (i=_thread_id; i<size; i+=_number_of_threads)   // Constant time prefix sum algorithm
      {     prefix(p,MPADD,&sum_,i);
            source_[i]=p;
      }
      return 0;
}
```

(B) The BASELINE version (global memory unit id is 0)

```
_main
    OP0     -4        ADD0  O0,R29  ST0   R30,A0   WB30  R29
    OP0     -8        OP1   8       OP2   ___main  ADD0  O0,R29 SUB1  R29,O1  ST0   R31,A0  JMPL  O2    WB29  A1
__BLOCK18
    OP0     176       OP1 __thread_id OP2 65535    ADD0  O1,R32 LD0   A0      SLE   M0,O2   TRAP  O0    WB3   O2    WB31  M0 WB1  AIC
    OP6     L3_LF0    BEQZ  O6
__BLOCK26
    OP0     _source_  OP1   __group_table_ WB5    O0    WB4   O1
L5_LF0
    OP0     _sum_     OP1   2       SHL0  R31,O1   ADD1  A0,R5  BMPADD0 R31,O0 WB6 O0      WB1   A0    WB1   A1    WB2   M0
    SMPADD0 R2,R6
    OMPADD0 R2,R6     WB2   M0
    ST0     R2,R1
    OP6     L5_LF0    LD0   R4      ADD5  R31,M0   SLE   A5,R3  BNEZ  O6      WB31  A5    WB1   AIC
__BLOCK41
L3_LF0
    OP0     180       TRAP  O0      WB1   R0
    OP0     _exit     JMPL  O0
```

(C) The LOCAL MEMORY -aware version (local memory unit version id is 0, global memory unit id is 1)

```
_main
    OP0     -4        ADD0  O0,R29  ST0   R30,A0   WB30  R29
    OP0     -8        OP1   8       OP2   ___main  ADD0  O0,R29 SUB1  R29,O1  ST0   R31,A0  JMPL  O2    WB29  A1
__BLOCK18
    OP0     176       OP1 __thread_id OP2 65535    ADD0  O1,R32 LD0   A0      SLE   M0,O2   TRAP  O0    WB3   O2    WB31  M0 WB1  AIC
    OP6     L3_LF0    BEQZ  O6
__BLOCK26
    OP0   _source_    OP1   __number_of_threads    ADD0  O1,R32 WB5   O0      WB4   A0
L5_LF0
    OP0     _sum_     OP1   2       SHL0  R31,O1   ADD1  A0,R5  BMPADD1 R31,O0 WB6 O0      WB1   A0    WB1   A1    WB2   M1
    SMPADD1 R2,R6
    OMPADD1 R2,R6     WB2   M1
    ST1     R2,R1     OP6   L5_LF0  LD0   R4       ADD5  R31,M0 SLE   A5,R3  BNEZ  O6      WB31  A5    WB1   AIC
__BLOCK41
L3_LF0
    OP0     180       TRAP  O0      WB1   R0
    OP0     _exit     JMPL  O0
```

Figure 4. Prefix as e and assembler programs for the baseline and proposed local memory -aware solutions. Local memory subinstructions are shown with **bold** typeface.

Benchmark	T_{base}	P_{base}	W_{base}	T_{prop}	P_{prop}	W_{prop}	N	Description
barrier	1	*N*	*N*	*N*	1	1	*T*	Perform a barrier synchronization for the threads in the group
block	1	*N*	*N*	*N*	1	1	*T*	Copy a table of N integers to another place in the memory
prefix	1	*N*	*N*	*N*	1	1	*T*	Compute an ordered multiprefix of a table of N integers
setup	1	*N*	*N*	*N*	1	1	*T*	Setup the local variables
update	1	*N*	*N*	*N*	1	1	*T*	Update the thread numbering for a synchronous construct

Table 1. Benchmarks used in evaluation.

Configuration		E4	E16	E64	C4	C16	C64
Memory system		baseline	baseline	baseline	proposed	proposed	proposed
Number of processors	P	4	16	64	4	16	64
Number of functional units	F	10	10	10	10	10	10
Number of threads per processor	T_p	512	512	512	512	512	512
Number of global memory units	M_g	1	1	1	1	1	1
Number of local memory units	M_g	0	0	0	1	1	1
Size of data memory (MB)	S_m	4	16	64	4	16	64

Table 2. CMP configurations used in the evaluation.

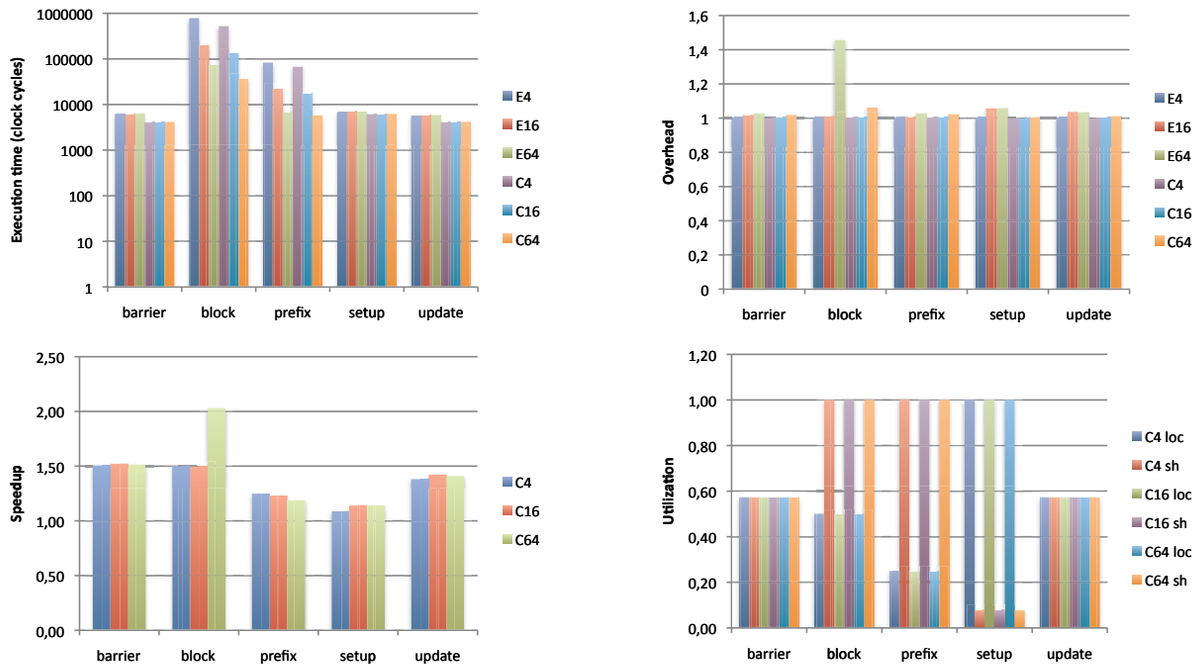


Figure 5. Execution time (top left), overhead with respect to ideal machine with the same instruction set (top right), achieved speedup (bottom left), and utilization of memory units (bottom right). Note that the problem size is proportional to the total number of threads in the architecture.

4.2 Compiling challenges

The compiling for the proposed architecture can be made similarly as for the baseline CESM. The challenge is to assign the correct memory operations to right memory units. This can be made similarly as in the virtual instruction-level optimization algorithm [Forsell03] but knowledge about the access space is needed. It should, however, be noted that this technique would provide only suboptimal results, since for the optimal reasons, full parallel alias analysis would be needed.

5. Evaluation

In order to evaluate the performance and utilization of the proposed local memory -aware technique, we applied it to the TOTAL ECLIPSE framework being developed at VTT [Forsell10].

5.1 Preliminary performance simulations

We measured the performance of the solutions outlined in Sections 2 and 3 by simulating execution of five benchmark problems representing important and widely used primitives of parallel computing and standard RTL (see Table 1) in six CESM CMP configurations having 4 to 64 512-threaded MBTAC processor cores featuring global memory only or both local and global memory aware memory systems in the PRAM mode (see Table 2).

The benchmark programs were compiled with the e-compiler, ec, with –O2 and –ilp optimizations on. The resulting programs were simulated with the IPSMSim tool modified for the proposed solution. The results of the simulations are shown in Figure 5.

5.2 Discussion

As expected, CMPs using the proposed memory system solution executed benchmark programs faster than the baseline CMPs assuming that both private and shared variables were used. The individual speedups ranged from 8.7% to 202% while the average speedups were 34%, 36% and 46% for C4, C16 and C64, respectively. The results are close to that of the 40% rule-of-thumb (characterizing typical benefit of an additional memory unit). The overheads with respect to a similar machine with ideal memory system were less than 1% in average in the proposed machine. This is only one fifth of that in the baseline machine. The main reason for this is reduced traffic in the inter-communication network since the private data accesses are targeted the local memories. The selected benchmark set showed quite different kinds of memory patterns, some providing equally high utilizations for both local and shared memory units, while two had more shared memory accesses than local memory accesses and one benchmark had more local memory accesses than shared memory accesses.

6. Conclusions

We have described adding/making use of local memories to CESM processors for storing thread-private data in the PRAM mode for performance and safety reasons. While this kind of solutions typically introduces a partitioning problem, most parallel computing languages make use of the concepts of shared and private data that provide natural orthogonal partitioning of data to global and local subspaces. According to our evaluation, the performance increases 40% in average while individual speedups range from 8.7% to 202%. The hardware implementation of the proposed solution is very simple since the CESM

architecture provides the local memories and memory units for the NUMA mode operation anyway.

Our future work includes implementing an FPGA prototype of the CESM making use of this technique. We will also study possibilities to implement compiler support for automatic assignment of right data to rights units/memories.

Acknowledgements

This work was supported by the REPLICA frontier research project of VTT.

References

[Abolhassan03] F. Abolhassan, R. Drefenstedt, J. Keller, W. Paul, D. Scheerer, On the Physical Design of PRAMs, Computer Journal 36, 8 (1993), 756-762.

[Alverson90] R. Alverson, D. Callahan, D. Cummings, B. Kolblenz, A. Porterfield, B. Smith, The Tera Computer System, Proceedings of the International Conference on Supercomputing, Association for Computing Machinery, New York, 1990, 1-6.

[Forsell02a] M. Forsell, A Scalable High-Performance Computing Solution for Network on Chips, IEEE Micro 22, 5 (September-October 2002), 46-55.

[Forsell02b] M. Forsell and V. Leppänen, Memory Module Structures for Shared Memory Simulation, In the Proceedings of the SSGRR-2002w, January 21-27, 2002, L'Aquila, Italy.

[Forsell02c] M. Forsell, Cacheless Instruction Fetch Mechanism for Multithreaded Processors, WSEAS Transactions on Communications 1, 1 (2002), 150-155.

[Forsell03] M. Forsell, Using Parallel Slackness for Extracting ILP from Sequential Threads, In the Proceedings of the SSGRR-2003s, July 28 - August 3, 2003, L'Aquila, Italy.

[Forsell05] M. Forsell, Realising constant time parallel algorithms with active memory modules, International Journal of Electronic Business 3, 3-4 (2005), 255-263.

[Forsell06] M. Forsell, Realizing Multioperations for Step Cached MP-SOCs, In the Proceedings of the International Symposium on System-on-Chip 2006 (SOC'06), November 14-16, 2006, Tampere, Finland.

[Forsell09] M. Forsell, Configurable Emulated Shared Memory Architecture for general purpose MP-SOCs and NOC regions, In the Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip, May 10-13, 2009, San Diego, USA, 163-172.

[Forsell10] M. Forsell, TOTAL ECLIPSE—An Efficient Architectural Realization of the Parallel Random Access Machine, In Parallel and Distributed Computing Edited by Alberto Ros, IN-TECH, Vienna, 2010, 39-64. (ISBN 978-953-307-057-5)

[Forsell11a] M. Forsell, A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads, International Journal of Networking and Computing 1, 1 (January 2011), 21-35.

[Forsell11b] M. Forsell, M. Penttonen and V. Leppänen, Cost of Sparse Mesh Layouts Supporting Throughput Computing, to appear in the Proceedings of EuroMicro Digital Systems Design 2011 (DSD'11), August 31-September 2, 2011, Oulu, Finland.

[Forsell11c] M. Forsell, Performance comparison of some shared memory organizations for 2D mesh-like NOCs, Microprocessors and Microsystems 35, 2 (March 2011), 274-284.

[Fortune78] S. Fortune and J. Wyllie, Parallelism in Random Access Machines, Proceedings of 10th ACM STOC, Association for Computing Machinery, New York, 1978, 114-118.

[Gajski83] D. Gajski, D. Kuck, D. Lawrie and A. Sameh, CEDAR—A Large Scale Multiprocessor, Proceedings of International Conference on Parallel Processing,1983, 524-529.

[Leppänen96] V. Leppänen, Studies on the realization of PRAM, Dissertation 3, Turku Centre for Computer Science, University of Turku, Turku, 1996.

[Pfister85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E.A. Melton, V. A. Norton and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, Proceedings of International Conference on Parallel Processing (1985), 764-771.

[Schwarz80] J. T. Schwarz, Ultracomputers, ACM Transactions on Programming Languages and Systems 2, 4 (1980), 484-521.

[Valiant90] L. G. Valiant, A Bridging Model for Parallel Computation, Communications of the ACM 33, 8 (1990), 103-111.

[Swan77] R. Swan, S. Fuller and D. Siewiorek, Cm*—A Modular Multiprocessor, In the Proceedings of NCC, 645-655, 1977.

[Vishkin11] U. Vishkin, Using Simple Abstraction to Reinvent Computing for Parallelism, Communications of the ACM 54, 1 (January 2011), 75-85.

On Maximizing Resource Utilization for Simultaneous Multi-Threading (SMT) Processors by Instruction Recalling*

Yilin Zhang Caleb Douglas Wei-Ming Lin
Department of Electrical and Computer Engineering
The University of Texas at San Antonio
San Antonio, TX 78249-0669, USA

Abstract— *Simultaneous multi-threading (SMT) has been a very popular design in improving resource utilization by sharing key datapath components among multiple independent threads. When critical resources are shared by multiple threads, to effective use of these resources proves to be the most important factor in fully exploiting the system potential. Allowing any of the threads to overwhelm these shared resources not only leads to unfair thread processing but may also result in severely degraded overall performance. How to prevent idling threads from clogging the critical resources in the pipeline becomes a must in sustaining system performance. In this paper, we show that, if one can manage to recall instructions of idling threads from Issue Queue (IQ), a very important shared resource in the SMT pipeline, the system performance is easily enhanced by a significant margin. An even more noteworthy feature about this technique is that the ensuing hardware overhead is very insignificant and its effect in clock timing also is negligible. The feature proposed in this paper can also be easily coupled with other advanced techniques employed in other stages of the SMT pipeline.*

Keywords: SMT, Instruction Recalling

1. Introduction

Simultaneous Multi-Threading (SMT) offers an improved mechanism to enhance the overall performance by exploiting Thread-Level Parallelism (TLP) [1], [2] to overcome the limited of Instruction-Level Parallelism (ILP), based on the traditional superscalar processors. The most common characteristic of SMT processors is the sharing of key datapath components among multiple independent threads, which ensures improved resource utilization. Subsequently, due to the sharing of resources, the amount of hardware required in an SMT system is significantly less than employing multiple superscalar machines while achieving similar performance.

Due to the requirement in resource sharing (those resource that are cost-wise better to share such as Issue Queue, functional units, etc.), these hardware components tend to remain busy in order to accommodate more instructions from all threads. Although allowing these instructions to share these resources ensures the full performance potential afforded by SMT [13], it tends to induce extra control complexities in managing the critical timing path and the processors cycle. To retain the exploitation of both TLP and ILP, a necessary solution must be introduced in order to minimize the complexity among these shared resources without affecting the ILP exploitation significantly. At the same time, an appropriate level of intelligence has to be incorporated into the resource sharing mechanism to ensure that threads share these components in the most efficient and fair manner.

Recently there have been many research activities targeted in improving SMT performance through modifying various stages in the pipeline. These include: improvement of thread co-scheduling by using probabilistic modelling for prediction in [8], avoiding register allocation by predicting transient values from branch misprediction in [3], packing instructions in issue queue to reduce delay and power consumption in [14], improving SMT fetching with an estimation of outstanding work in the system for each thread in [18], early deallocation of registers in association with cache misses in [12], dynamically reconfigurable cache design for better IPC in [6], a modified fetch policy with adaptive resource partitioning in [19], and another fetch policy by considering memory-level parallelism in [7]. None of these techniques specifically addresses the contention in the issuing stage and most come with a significant requirement in extra hardware to implement the desired intelligence.

In this paper, we show that, if one can manage to recall instructions of idling threads from Issue Queue (IQ), a very important shared resource, potential clogging from threads that are temporarily idle can be drastically relieved, and the system performance is easily enhanced by a significant margin. An even more noteworthy feature about this technique is that the ensuing hardware overhead is very insignificant and its effect in clock timing also is negligible. This technique can also be paired with all the techniques aforementioned in the literature which emphasize on other stages of pipeline to further improvement the performance.

*This material is based in part upon work supported by the National Science Foundation under Grant Number HRD 0932339. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

2. Instruction Dispatch

There have been many different terminologies adopted for instruction pipelining stages (e.g. issue, dispatch, etc.) in a superscalar system. Throughout this paper, we choose to adopt the terminologies used by most SMT articles, which is also showed in Figure 1.

In a typical single-thread superscalar system, instructions are "dispatched" from ROB into the reservation stations (either centralized or functional unit-specific) whenever space is available and then "issued" to the corresponding functional unit whenever the issuing conditions are met, i.e. operands are ready and the requested functional unit becomes available. However, in a basic SMT system, each thread has its own ROB and instructions from these thread-specific ROBs have to "compete" for a shared Issue Queue (IQ) through a dispatching scheduling algorithm. This IQ can be considered as Centralized Research Station not only shared among the functional units but also shared among the threads in real time. A basic functional block diagram of this basic design is depicted in Figure 1.

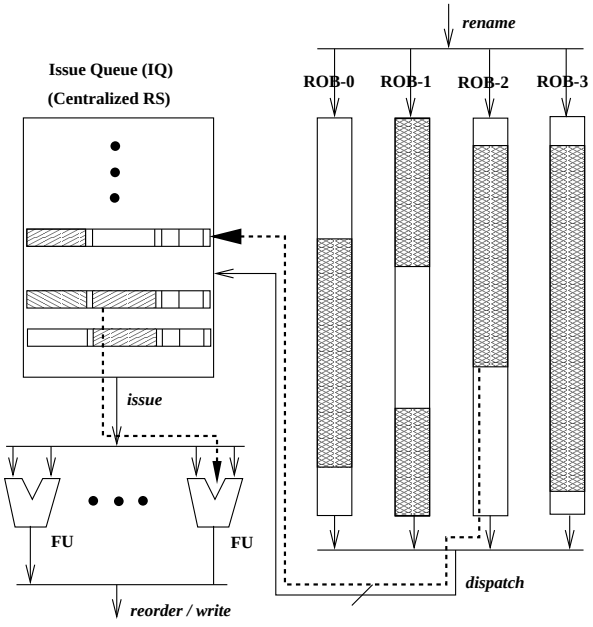


Fig. 1: A Simple Four-Thread SMT Instruction Processing Flow

Due to the significantly large size of each IQ entry, the number of entries in this shared resource usually is much smaller than the number of ROB entries. Having its output sent into a tightly shared resource, the instruction dispatch stage is considered one of the most critical stages that dearly affect the overall system performance.There have been many non-adaptive dispatching algorithms proposed, including simple round-robin, within-clock-cycle-round robin [5], and some other techniques intend to exploit ILP by scheduling instructions to be dispatched from ROB, which relies on simple instruction-level readiness to reduce the instructions' waiting time in IQ [15]. To better utilize the shared IQ entries, instructions from different threads should be given different priorities depending on the threads' "activeness" in real time. Threads have been shown to demonstrate various types of transient behaviors throughout their life span, including stretches of time durations with fluctuating ILP. Instructions from thread(s) of lower ILP (or simply "slower" in instruction issuing) would clog the IQ if they are allowed to continue to be dispatched into IQ. On the other hand, instructions from higher ILP (or simply "faster" in instruction issuing) should be given a higher priority in utilizing the IQ. There have not been many research results on how to share the IQ in a more time-adaptive manner allowing different threads to utilize (occupy) the IQ in an "on-demand" basis. In [11], an adaptive technique is proposed to allow each of the ROBs to be "partially" used to accommodate threads that are not as active, in which "activeness" of a thread is based on a ratio between number of issue-bound and commit-bound instructions in a thread's ROB. Other more advanced scheduling techniques, such as the one presented in [20], combine more information from different stages in the pipeline to optimize the scheduling/dispatching result, albeit requiring significantly more hardware and control logic.

Our analysis shows that the activeness of a thread can be fluctuating very unexpectedly in time due to cache misses, branch miss-predictions, write port latencies, etc. A thread that has been active can suddenly become "inactive" and stay "idle" in the pipeline for a long duration of time. To make it worse, these threads that have been just recently more active (than other threads) tend to occupy more shared resources (e.g, the IQ) than others. When these threads suddenly become inactive, the shared resources typically cannot be released for other threads to use, mainly due to various system or operating limitations inhibiting it from doing so and consequently limit the potential performance. Besides, to further improve the system resource utilization at this level of high-speed instruction processing, one cannot afford employing any design that involves too much intelligence for real-time implementation.

In this paper, we choose to retain the per-thread ROBs without any sharing among them, and rely on a simple scheduling algorithm in dispatching instructions from different threads. The basis for performance comparison in this paper will be with a more advanced round-robin dispatch scheduler in which all threads take turn dispatching at most one instruction in its own turn during a clock cycle until the bandwidth is consumed. The average and Harmonic mean of IPC will be compared to see the improvement.

3. Simulation Environment

The simulation environment including the simulator and the workloads used for our simulations are first described in this section.

3.1 Simulator

We used M-Sim [10], a multi-threaded micro architectural simulation environment model, to estimate the overall performance of the proposed scheme. M-sim includes accurate models of the pipeline structures such as explicit register renaming, concurrent execution of multiple threads, detailed power estimation using Wattch framework [16], separate Reorder Buffer, and Load-Store Queue (LSQ) which are necessary for an SMT model. The Issue Queue and execution functional units are shared among the threads, but register files and branch predictor is exclusive to each thread. The detailed processor's configuration is shown in Table 1.

Parameter	Configuration
Machine Width	8 wide fetch/dispatch/issue/commit
L/S Queue size	48-entry Load/Store queue
IFQ size	32-entry Instruction Fetch queue
ROB& IQ size	as mentioned
Function Units & Latency (total/issue)	8 Int Add (1/1) 4 Int Mult (3/1) / Div (20/19) 4 Load/Store (2/1), 8 FP Add (2) 4 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
Physical registers	256 integer and floating point
L1 I-cache	64KB, 2-way set-associative 128 byte line
L1 D-cache	32 KB, 4-way set-associative 256 byte line
L2 Cache unified	2 MB, 8-way set-associative 512 byte line
BTB	2048 entry, 2-way set-associative
Branch Predictor	2K entry gShare 10-bit global history per thread
Pipeline Structure	5-stage front-end (fetch-dispatch) scheduling (for register file access: 2 stages, execution, write back, commit)
Memory	64 bit wide, 200 cycles access latency

Table 1: Configuration of the Simulated Processor

3.2 Workloads

For multi-threaded workloads, we use the mixed SPEC CPU 2000 benchmark suite [4], [9], [16] based on ILP classification. Each of the benchmarks is initialized in accordance with the procedure mentioned in Simpoints tool [17] and then up to 100 million instructions are simulated in a simple scalar environment. As shown in Table 2, three types of ILPs - low ILP (memory bound), medium ILP and high ILP (execution bound) - are identified, and 12 mixes of multi-threaded workloads are used based on different combinations of ILP types.

Mix	Benchmarks	Classification (ILP)		
		Low	Med	High
Mix 1	swim, locus, galgel, twolf	0	0	4
Mix 2	locus, galgel, twolf, applu	0	0	4
Mix 3	equake, vpr, mesa, ammp	0	4	0
Mix 4	vpr, mesa, ammp, gap	0	4	0
Mix 5	gcc, perlbmk, crafty, mgrid	4	0	0
Mix 6	crafty, mgrid, apsi, bzip2	4	0	0
Mix 7	crafty, mgrid, lucas, galgel	2	0	2
Mix 8	gcc, perlbmk, twolf, applu	2	0	2
Mix 9	apsi, bzip2, equake, vpr	2	2	0
Mix 10	mgrid, apsi, ammp, gap	2	2	0
Mix 11	swim, lucas, vpr, mesa	0	2	2
Mix 12	twolf, applu, ammp, gap	0	2	2

Table 2: Simulated Multi-threaded Workload

4. Proposed Method

In a typical SMT system, a shared IQ (Issue Queue), which is used as a set of centralized reservation stations for all functional units, is required to hold all the necessary information for instructions. Thus, further increasing the number of entries in the shared IQ usually is hampered by the cost factor, and therefore the utilization of these limited resources becomes very critical to the overall system performance.

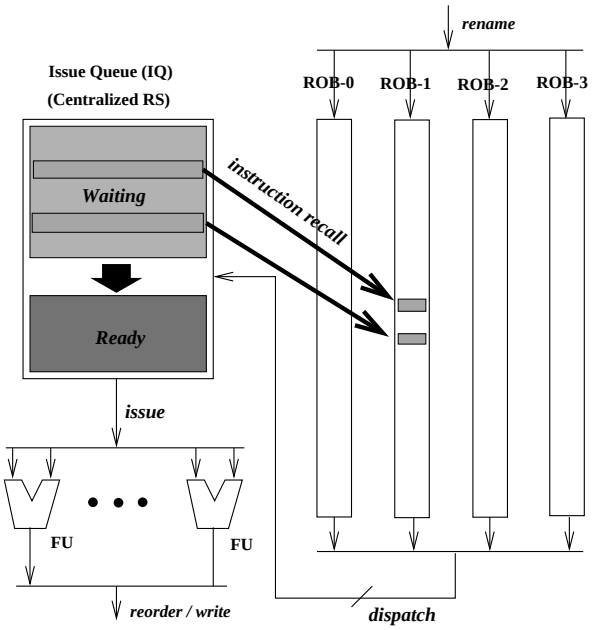


Fig. 2: Instruction Recalling

Once an IQ slot is allocated to an instruction, all known architectures will allow it to stay in IQ until it eventually becomes ready for execution, even it may take hundreds of clock cycles. Our proposed technique instead will recall such an instruction if the owning thread shows a sign of inactiveness. Figure 2 shows a simple illustration of such instruction recalling. Since a dispatched instruction recalled

is still in ROB, recalling it actually involves setting/resetting a flag in each of the two entities, one in the IQ entry and the other one in the original ROB entry. This process essentially re-declares the IQ slot to become available (empty) and resets the instruction in ROB to become "yet-to-dispatch". To determine whether a thread is "inactive" may require some extra intelligence incorporated into the system. Any of the operands that have become ready after the instruction was dispatched can be easily re-established (read) again when the instruction is dispatched again.

4.1 Motivation

The proposed technique is based on the conjectures that, if IQ is not properly managed, instructions do stall in IQ for a long period of time, its imbalanced occupancy scenario among threads does exist. This section will be devoted to the discussion to support these conjectures.

4.1.1 IQ Occupancy Latency Analysis

We first look into behavior of instructions after they are dispatched into the IQ; namely, how long they stay in IQ and how long it takes for them to become ready for issuing to the corresponding functional units. Figure 3 shows the cumulative percentage of instructions that spend in IQ for at least the given number of clock cycles, where R is used to denote the number of ROB entries and q is the size of IQ used. As Figure 3 shows, close of 30% of instructions

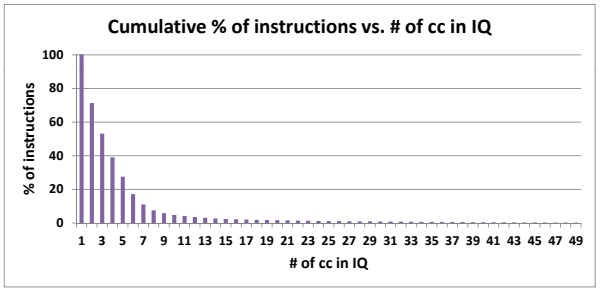


Fig. 3: IQ Latency Cumulative Distribution with $(R, q) = (96, 24)$

encounter an IQ latency of at least 5 clock cycles, and there are a significant number of instructions that get stuck in IQ for a long period of time, e.g., about 1% of instructions spend more than 30 clock cycles in the IQ. These IQ slots could have been better used if these instructions can be recalled after their latency exceeds a certain threshold.

4.1.2 Per-Thread IQ Occupancy Analysis

Another scenario we want to look into is how exactly the IQ may be dominated by a single thread among the threads in SMT processing. Figure 4 shows the average percentage of time (out of the 12 mixes used) that at least one thread is occupying at least the given number of IQ entries. This

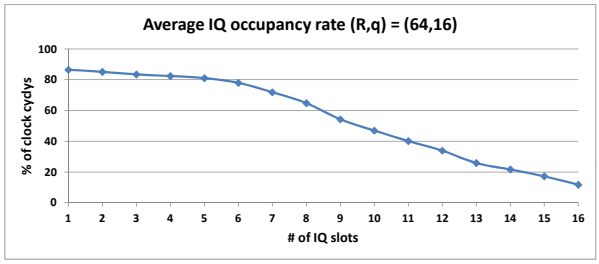


Fig. 4: Average IQ Occupancy Rate

clearly shows that on the average in over 60% of clock cycles there is at least one thread occupying more than 50% of IQ slots, and three quarters of resources are tied up by one thread in over one third of time. Once such a dominating thread somehow becomes "inactive", the consequence can be significant.

4.2 IQ Instruction Recalling

One straightforward method in effectively using the IQ is to allow the slots that have been occupied by stagnant instructions to be freed for other instructions. To "correctly" classify whether an instruction (or more specifically its owning thread) is considered "stagnant" may require an intelligence too expensive or too execution-time-critical to implement. We choose to simply set a time threshold for a thread to be considered "inactive" (or "blocked") when this thread has not (1) any of its instructions in IQ become "ready" and (2) any of its instructions committed during this time period. The complete algorithm is depicted in Figure 5. This threshold is referred to as "Cycles-of-Stall-to-Blocked" (CSB). Such a condition is chosen due to the consideration that a blocked thread usually displays one or both of the two symptoms. In our algorithm, once a thread reaches this threshold value, all its instructions in IQ that are not ready for issuing will be recalled. This condition-checking step is shown in the flowchart as "stall_cc==CSB?". As aforementioned, the recalling process of an instruction involves two simple logical operations: resetting the "free" flag of the IQ slot back to "1" (yes), and resetting the "dispatched" flag of the instruction in ROB to "0" (no). This thread, once recalled, will not be allowed to dispatch again until (1) it starts resuming instruction committing, or (2) a time-out is reached. If a thread so-declared "blocked" starts to commit instructions again, then most likely the condition that rendered this thread stagnant is resolved. On the other hand, there may exist situations that a thread simply does not have any instruction dispatched in the system after the recalling and thus will not be able to resume the active status by committing an instruction. In such a case, this time-out is a necessary mechanism for the thread to have a chance to become active again, even it may not be ready yet. Such a time-out value will be referred to as the "Cycles-of-Blocked-to-Reactivate" (CBR) value. In the flowchart this time-out is

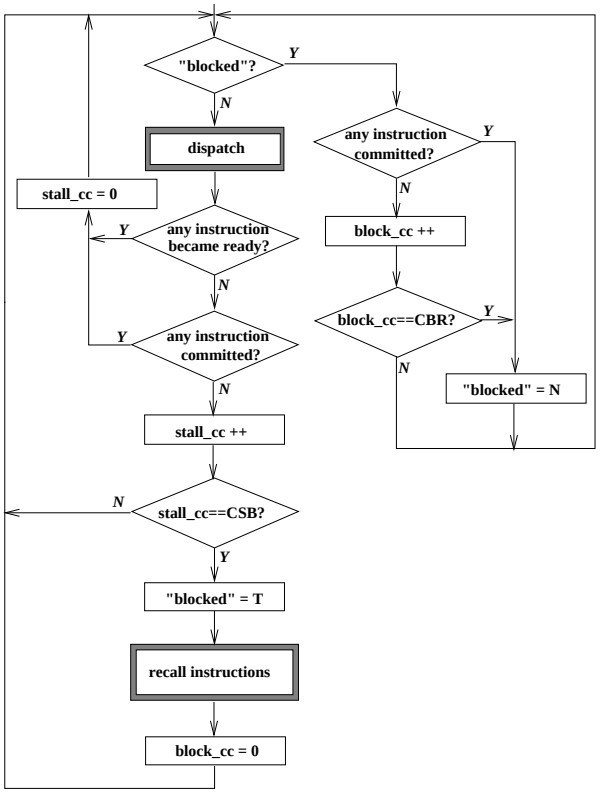


Fig. 5: Instruction Recalling Algorithm

checked with the "block_cc==CBR?" condition.

Note that the selection of CSB value and CBR value greatly influences the effect of this technique in improving the overall performance. A too small CSB value will lead to unnecessary recalling which will be a waste of resource, while a too large CSB value will diminish the effect of this technique. Similarly, the adoption of a proper CBR also may have a significant impact on the performance. A too small CBR will dampens the effect of recalling and a too large CBR will jeopardize specific thread from effectively using resource. One should suspect there should be a pattern of matching CBR and CSB leading to best resource utilization, which will be discussed in our simulation results.

5. Simulation Results

Based on the simulation environment and workloads described in Section 3, our proposed technique is tested compared to the default dispatching system. An average of overall performance from 12 mixes as described in Section 3.2 is used for presentations. Figure 6 shows the average IPC improvement percentage values when different CSB values are applied, with the reactivation delay CBR fixed at 20 clock cycles. Three different combinations of ROB size (R) and IQ size (q): $(R, q) = (128, 32)$, $(96, 24)$ and $(64, 16)$ are tested. From this chart, one can tell among the three (R, q) combinations the one with the smallest IQ

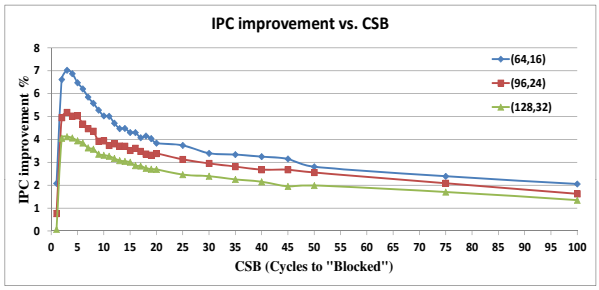


Fig. 6: IPC Performance Improvement by Using Instruction Recalling

size leads to the best improvement, up to 7% in average IPC. The larger the IQ size is the smaller the improvement is obtained. This scenario can be easily explained by the more critical effect of our instruction-recalling technique on a smaller IQ than on a larger one. Also note that no matter what combination of (R, q) is used, the best improvement in IPC happens when the CSB values is set to about 3 or 4. The fact that the "optimal" CSB value is so small indicates that the benefit from recalling more instructions (with a small CSB value), even though these instructions may not be exactly "blocked", may clearly outweigh the loss in dispatching capacity utilization due to the recalling. When the CSB value is set to an even smaller value, the effect of loss quickly diminishes the benefits.

Comparison of individual IPC values for each mix between the default and the proposed techniques is presented in Figure 7, with $(R, q) = (64, 16)$ and (CSB, CBR) =(4,20). From these results, the proposed technique consistently leads

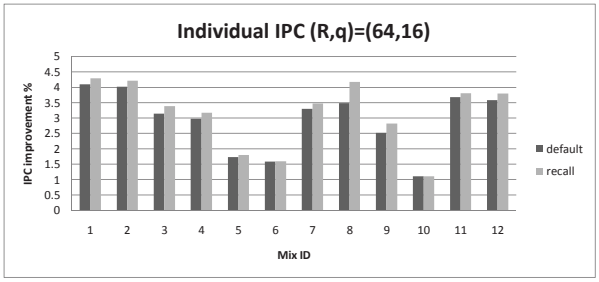


Fig. 7: IPC Performance Comparison for Each Mix

to improved IPC over the default technique throughout all mixes and all combinations of queue sizes, with improvement percentage up to 20% on some mixes.

To further demonstrate that overall IPC improvement achieved by the proposed technique does not jeopardize the fairness of threads' execution due to the recalling process, Figure 8 shows the harmonic mean IPC values for all mixes between the default and proposed techniques when $(R, q) = (64, 16)$. The formula used to calculate the harmonic mean follows the standard definition in $H = n / \sum_i^n \frac{1}{IPC_i}$, where n is the number of threads in this case. From these, one

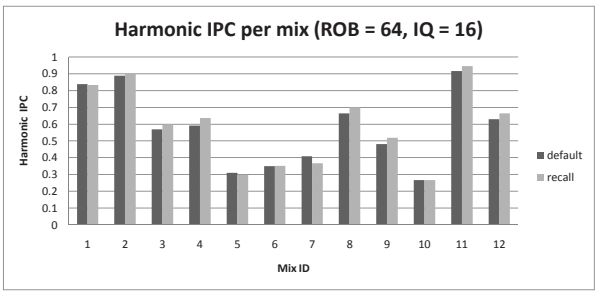


Fig. 8: Harmonic IPC Performance Comparison for Each Mix

can tell that the improvement on the overall IPC from the proposed recalling process not only does not lead to any imbalance among the threads but mostly incurs positive effect (up to 8% in improvement) in raising individual IPC for all threads.

From Figure 6 we can tell that the impact of setting a proper CSB value and it shows that the optimal CSB value stays very consistent throughout different combinations of buffer sizes, (R, q). Figure 9 shows the impact of CBR on overall performance when CSB is fixed. When the CSB

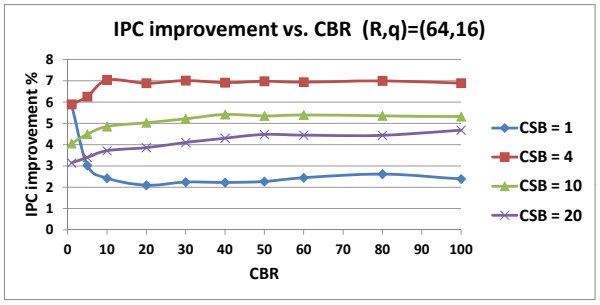


Fig. 9: IPC Performance versus CBR

value is set to the optimal value of 4, the IPC performance stays relatively unchanged once the CBR is set to be more than 10. It is reasonable because most threads can reactive themselves by resuming committing, thus a large CBR affects very little on the performance. However, if the CSB is set to a very small value, e.g., 1 as shown in the figure, the IPC drastically suffers once the CBR is set to a value higher than 1. This can be explained by the fact that with such a small CSB, a lot of instructions are unnecessarily recalled, and if these instructions are not re-dispatched soon again (under a larger CBR value), then the performance degrades easily. From this result, one can tell that the selection of CSB is much more critical than selecting CBR as long as the latter is not set to a value too small.

One analysis that can somewhat reveal the true impact of the proposed recalling technique is on the percentage of all executed instructions that have been recalled. Figure 10 shows this percentage value when a different CSB threshold is adopted. One can see that less than 3% of dispatched

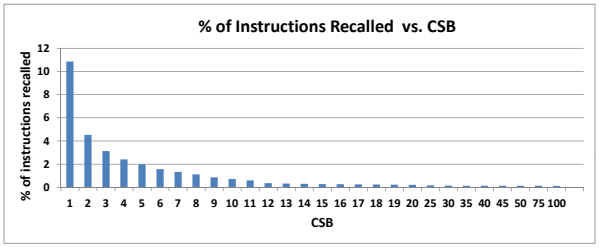


Fig. 10: Percentages of Instruction Recalled vs. CSB

instructions are recalled when the CSB value is set to at least 4, and only about 11% of instructions are recalled when CSB is set to 1.

Out of all the instructions dispatched that are recalled, some of them actually are of the same instructions that are recalled multiple times. The next analysis is to see, among all the instructions that are recalled, how often an instruction is recalled multiple times. Figure 11 shows the percentage of instructions that are recalled a given number of times with different (CSB,CBR) patterns. The figure

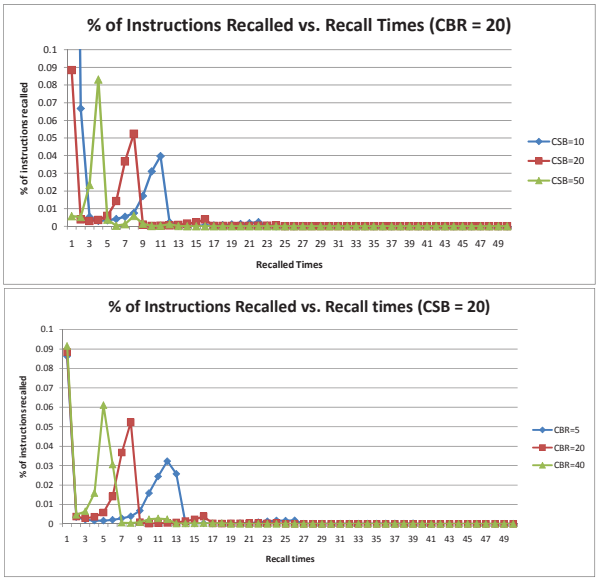


Fig. 11: Percentages of Instruction Recalled vs. Recalled Times under Different (CSB,CBR) Patterns

shows a very interesting pattern in having a clear spike indicating that some significant number of instructions are recalled a somewhat specific multiple times, and this specific number depends on both the CSB and CBR value. A close investigation into these patterns reveals that

$$p \times (\text{CSB} + \text{CBR}) \approx C \tag{1}$$

where p is where the peak appears at, when multiplied by the total cycles for each repeated recall, the product is

approximated by a constant C, in the range of 250 to 300 in all cases.This somewhat constant value corresponds very close to the cache miss penalty. This indicates that this peak corresponds to the group of instructions that were delayed by a cache miss situation. Based on this observation, extra intelligence can be further incorporated into the system to specifically detect the occurrence of a "blocked" situation due to cache miss and subsequently set a proper CBR value to prevent redundant repeated recalls of the same instructions from the thread.

To further analyze the effect of recalling on the IQ occupancy rate, the average per-thread IQ occupancy analysis is performed again when the recalling technique is applied, similar to the one shown in Figure 4. Figure 12 displays the comparison. Note that after the recalling technique is

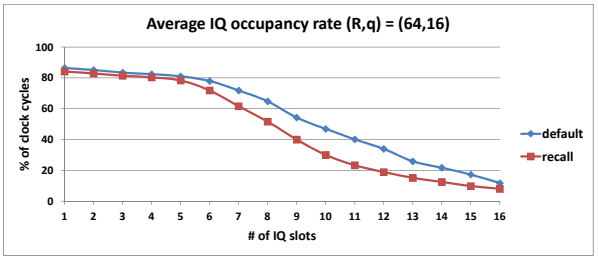


Fig. 12: IQ Occupancy Rate Comparison

applied, the "blocking" from a thread in IQ is not as severe – there was about 65% of clock cycles in the default case that at least one thread takes up more than half of the IQ slots, and, with the recalling, only about 51% of time such a situation happens. Three quarters of slots were occupied by a thread in 34% of time, while it is reduced to about 19% by the recalling method. This analysis clearly shows the effect of instruction recalling in relieving the IQ blocking.

6. Conclusion

This paper clearly demonstrated that utilization of resources shared among the threads in an SMT system could significantly affect the overall performance. By recalling idling instructions from the critically shared resources, overall performance can be vastly improved without sacrificing fairness of execution among the threads. In addition, such an improvement is achieved without having to invest much extra hardware or imposing extra constraints on the clock rate. Incorporating further intelligence into such a recalling process will likely improve the performance more while one would need to justify the potential in leading to excessive hardware requirement.

References

[1] D. Tullsen et al., "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. Intl Symp. Computer Architecture*, 1995.

[2] D. Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. Intl Symp. Computer Architecture*, 1996.

[3] D. Balkan, J. Sharkey, D. Ponomarev and K. GhoseSPARTAN, "Speculative Avoidance of Register Allocations to Transient Values for Performance and Energy Efficiency," *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, September 2006.

[4] D. Burger, T. Austin. "The SimpleScalar tool set: Version 2.0." *Tech. Report*, Dept. of CS, Univ. of Wisconsin-Madison, June 1997 and documentation for all Simplescalar releases.

[5] M. Debnath, B. Lee, and W.-M. Lin, "Prioritized Out-of-Order Instruction Dispatching Techniques for Simultaneous Multi-Threading (SMT) Processors", WIP session of 30th IEEE Real-Time Systems Symposium (RTTS), Dec. 2009.

[6] J. Daz, J. .Hidalgo, F. Fernndez, O. Garnica and S. Lpez, "Improving SMT Performance: an Application of Genetic Algorithms to Configure Resizable Caches," *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, July 2009.

[7] S. Eyerman and L. Eeckhout, "Memory-Level Parallelism Aware Fetch Policies for Simultaneous Multithreading Processors," *Transactions on Architecture and Code Optimization (TACO)*, Volume 6 Issue 1, March 2009.

[8] S. Eyerman and L. Eeckhout, "Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling," *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, March 2010.

[9] J. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *Transactions of IEEE Computer*, 33(7):2835, July 2000.

[10] J. Sharkey, "M-Sim: A Flexible, Multi-threaded Simulation Environment." *Tech. Report CS-TR-05-DP1*, Department of Computer Science, SUNY Binghamton, 2005.

[11] J. Sharkey, D. Balkan and D. Ponomarev, "Adaptive Reorder Buffers for SMT Processors", *the 15th international Conference on Parallel Architectures and Compilation Techniques*, 2006.

[12] J. Sharkey, J. Loew and D. Ponomarev, "Reducing Register Pressure in SMT Processors through L2-Miss-Driven Early Register Release," *Transactions on Architecture and Code Optimization (TACO)*, Volume 5 Issue 3, November 2008

[13] J. Sharkey and D. Ponomarev, "Efficient Instruction Schedulers for SMT Processors," *Proc. 12th Intl Symp. High Performance Computer Architecture (HPCA)*, 2006.

[14] J. Sharkey, D. Ponomarev, K. Ghose and O. Ergin, "Instruction Packing: Toward Fast And Energy-Efficient Instruction Scheduling," *Transactions on Architecture and Code Optimization (TACO)*, Volume 3 Issue 2, June 2006.

[15] J. Sharkey and D. Ponomarev, "Exploiting Operand Availability for Efficient Simultaneous Multithreading," *IEEE Transactions on Computers*, vol. 56, no. 2, February 2007.

[16] T. Sherwood, et al. "Automatically Characterizing Large Scale Program Behavior," *Proc. ASPLOS*, 2002.

[17] Standard Performance Evaluation Corporation (SPEC) website, http://www.spec.org/

[18] H. Vandierendonck and A. SeznecManaging, "SMT Resource Usage through Speculative Instruction Window Weighting," *Transactions on Architecture and Code Optimization (TACO)*, Volume 8 Issue3. November, 2008.

[19] H. Wang, I. Koren, and C. Krishna, "An Adaptive Resource Partitioning Algorithm for SMT Processors," *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[20] H. Wang, R. Sangireddy, "Optimizing Instruction Scheduling through Combined In-Order and O-O-O Execution in SMT Processors," *IEEE Transactions On Parallel And Distributed Systems*, vol. 20, no. 3, pp. 389-403, March 2009.

THE BACKGROUND AND IMPORTANCE OF EXPLOITING MULTIPLE CORES: A CASE STUDY IN NEUROPHYSIOLOGICAL VISUALIZATION

Roy Tucker, Nigel Barlow and Liz Stuart
The Visualization Lab, School of Computing & Mathematics, University of Plymouth, Plymouth, UK
roy.tucker@plymouth.ac.uk(contact author), nigel.barlow@plymouth.ac.uk, liz.stuart@plymouth.ac.uk

Keywords: Concurrency, Parallel Computation, Multithreading, Massive datasets, Visualization

Abstract: *This paper reviews the history and current use of multi-threading in software development. Over the last decade, there has been a complete paradigm shift in computer hardware. Currently, increased computing power is supplied using an increasing number of core processors. This paradigm shift has led to the development of higher level programming constructs and frameworks in many popular languages. Therefore, it is clear that both the hardware and software of the future are going to be based heavily upon concurrency. Despite this, software developers are still resistant to embrace multi-threading. This resistance is understandable due to the complexity of coding concurrency. However, this paper proposes that Concurrency is no longer an option but a necessity. In conclusion, the paper presents a case study based on the visualization of large quantities of Neurophysiological data. This application was developed using the Java concurrency framework.*

1. Introduction

During the early days of Computing, there was a considerable difference between the operating speed of CPUs and the speed of connected peripherals. Back then, CPU time was expensive, thus it was highly inefficient to pause the execution of a program to enable the execution of a slower input/output peripheral.

The first attempt to address this problem was the LEO III (Lyons Electronic Office) developed in 1961. LEO III employed the first multiprogramming system [1] which enabled a batch of programs to be loaded into the CPU simultaneously thereby essentially queuing for CPU time. In this system, the first program would execute until it reached an instruction which required the use of a peripheral device. At this point, the context of the first program would be stored thereby enabling the next program to execute. Subsequently, the use of CPU time was maximised. The main limitation of this system was that it required multiple programs to maintain this level of CPU usage. Nowadays this would be recognised as a problem of granularity, the executing units were too large to maximise the CPU usage.

The limitations of multiprogramming became obvious as computer systems moved from batch processing to interactive use. Multiprogramming was not capable of delivering well designed systems. One of the key benchmarks for well-designed systems was the set of "Golden rules" defined by Shneiderman [2]. These rules were, and still are, widely adopted throughout the industry. They emphasise the importance of providing "informative feedback" to the user.

Initial attempts to provide interactive feedback to users involved the co-operation of software developers. This was known as co-operative multitasking. This relied on developers ensuring their applications yielded CPU time to other applications.

Initially, this approach was successful. The earlier versions of Windows (prior to Windows 95) and the Mac operating system (prior to MAC OS X) employed co-operative multitasking [3, 4]. Whilst co-operative multitasking was increasingly deployed as a multitasking solution, another option called pre-emptive multitasking evolved.

In contrast, pre-emptive multitasking paradigm provides slices of CPU time to each of the executing processes. Effectively, this enforces the sharing of the CPU time. This implicit guarantee of CPU time enables developers to provide user with well- designed systems capable of proving "informative feedback" quickly. In 1969, this approach was selected for use in the UNIX operating system. It is standard in UNIX and its derived operating systems [5].

By the mid 1990's, Microsoft Windows had adopted the pre-emptive multitasking system incorporating it into both Windows NT and Windows 95. Apple Inc. followed suit with the MAC OS 9.x, released in October 1999.

2. The Pre-emptive Model

In the pre-emptive model, processes are separated into two categories:

- I/O Bound processes where the total computation time to complete the process is limited by the speed at which data can be requested and delivered [6]. Typically processes that make long read / write operations to disk would be I/O Bound and
- CPU Bound processes where the total computation time to complete the process is limited by the operating speed of the CPU. Typically a process that primarily 'crunches numbers' will be CPU bound.

This blocking mechanism enables the CPU time being consumed by these "waiting" processes to be re-allocated to processes in the CPU bound category. This continues until an interrupt signals to the I/O bound process that the blocked process can proceed.

As this pre-emptive model evolved, programmers began to develop applications as a collection of interacting (co-operating) processes. In turn, this raised the issue of how to efficiently share data between multiple processes. As originally conceived, a process ran in its own protected memory space isolated from other processes. However, this was not conducive to data sharing. The solution was that co-operating processes should share the same memory space. This approach would become known as multi-threading with multiple 'threads of execution' sharing a single processes memory space.

2.1 The paradigm shift from Moore's Law to Amdahl's law

Moore's Law states that the power of computer processing would double approximately every two years. Since its formulation in 1965, this law has provided a reliable guide to the growth of computing power. This predictable growth in computing power has been quietly exploited by software engineers worldwide. Until now, developers have enjoyed ever faster performance from their software simply by updating to newer hardware. However, it is proposed [7] that Moore's Law cannot be sustained as physical limitations for miniaturisation are encountered. Conversely, it is argued that advancing technologies will enable the law to survive far into the future [8].

Regardless of these opposing opinions, developers must consider how current and future technologies will deliver computing power. It is clear that hardware manufacturers have moved to the idea of delivering computer power through multi-core systems. Almost all forms of computer now employ multi core hardware as standard. This ranges from the mobile phone through to desktop computers of major research projects. Whilst considering highly compute intensive operations such as weather forecasting, computing power is now delivered by massively parallel super computers, grid computing and the opening of graphics processing cores for general non-graphical use (for example using Nvidia's CUDA) is becoming more widespread in research. Yet, it is still not enough.

Consequently, another law is now dominating the current expansion of computer processing and throughput. This is Amdahl's law which describes the performance increases that can be achieved through the parallelisation of software. Essentially a program is divided into two portions, the parallelisable and the sequential components. Additional compute cores will improve the execution speed of the parallel component of the program but these will have no effect on the sequential component [9].

2.2 Recent developments in hardware

In the last decade computer power has expanded based on the development of multithreaded execution architectures as well as the delivery of additional power using multicore systems. However, this increased processing capability can only be realised when software developers change their programming styles to exploit this new paradigm. Fundamentally, the latest hardware advances are completely dependent on the understanding and adoption of these new techniques by software developers. David Stewart CEO of CriticalBlue and chairperson for the Multicore Programming Practices (MPP) working group comments on this situation stating that "There's capability in (multicore) platforms which is not being utilized or not being optimized by the software development community" [10].

Developers are often deterred due to the difficulty of using threads and locks to control access to shared memory. Goetz [11], a key developer of the Java languages Concurrency Framework, states "writing correct programs is hard; writing correct concurrent programs is harder." Indeed, this style of programming has "a well-deserved reputation for introducing bugs that are difficult to find and fix." [10]. Even Apple Inc. dismisses it stating that "the dominant model for concurrent programming - threads and locks - is too difficult to be worth the effort for most applications." [12]

Despite the difficulties Goetz notes that "threads are the easiest way to tap the computing power of multiprocessor systems". Furthermore, "as processor counts increase, exploiting concurrency effectively will only become more important" [11]. Therefore if the true power of multicore systems is to be realised, it is essential that the next generation of developers will have the tools and the training to address the difficulties of concurrent programming.

2.3 Recent developments in software

Over the last few years several mainstream programming languages have evolved to address the tools needed for concurrent programming. These tools provide

concurrency frameworks that allow developers to work with abstract concepts rather than the lower level threads and locks.

This trend is likely to continue. Intel is currently experimenting with the Intel Array Building Blocks (Intel® ArBB) framework that will integrate with standard C++ applications without any compiler specific extensions [13]. This would remove a major compatibility hurdle to developing C++ parallel code.

Table 1 shows a selection of the major coding languages that have released a concurrency framework over the last decade. Stack-less Python represents a significant fork of the language that has enjoyed commercial success and development support (via CCP Games Inc). The Java Concurrency Utilities merged a concurrency framework with cross platform support to produce a flexible and widely deployable solution. Microsoft's Task Parallel Library went a step further seeking the same goal for a whole range of languages supported by the .NET framework. While all of these were high level languages, Intel have addressed the lower level C++ language. Note that they too are now experimenting with a new more general solution than the existing Threading Building Blocks (TBB).

The development of frameworks to remove the complexity of task and thread management is essential to promote multi-threaded computation and ensure it is accessible to the software development community. This is very helpful. However, software engineers must also be able to able to design, debug, validate and optimise code. Tools to support all these area of software development are at an early stage of their development. Areas such as debugging and validation are challenging. Patterson [14] states that multi-threaded code is well known to be notoriously difficult to debug and validate. The non-deterministic behaviour of concurrent code and the dangers of deadlock as well as race conditions are well understood. Despite these challenges, the tools to detect such errors during development are only now beginning to emerge.

For example, it was 2008 when Intel announced the development of a dedicated package of software engineering tools called the Intel® Parallel Studio suite [15]. The package was released to the development community in May 2009 [16]. With such tools only just beginning to emerge most developers remain uncertain how to validate and debug parallel code beyond repeated testing and code inspections. Issues of code optimisation, identification of parallelisable code and its long term maintenance are seldom considered by most developers.

3. Future Trends

Over the next decade, parallelism in software development will become more important both in research and in commerce applications. Indeed, the development company, CCP based in Iceland, is already crediting its commercial success to Stackless Python [17]. With the preference for multicore architectures firmly entrenched with manufacturers, the demand for software to exploit its power is unavoidable. However, this raises one of the primary issues, namely that of training software developers to exploit the hardware capability. Many of the current generation of programmers seldom consider parallel execution of code. This is understandable as it has mirrored the underlying hardware (subject to processor time slicing providing an illusion of parallel execution).

However, the next generation of developers must embrace the harder task of parallel code development as 'standard practice'. To achieve this training programs both in universities and industry must to be updated to emphasize parallel coding principles and to introduce the software engineering tools that will support the design, implementation, validation and tuning of parallel programs.

3.1 Power and Environmental issues

Power has traditionally been seen as being in abundant supply, but this view is changing. The emergence of cloud computing has led to the formation of warehouse sized data centres that "are now consuming more energy than heavy manufacturing in the United States" [18]. Subsequently, this has led to political pressures (in light of carbon emission targets and taxes) to reduce energy consumption and waste.

Table 1: Concurrency Frameworks for major coding languages

Language	Frame-work	Re-lease Date	Source
Java	JSR 166: Concurrency Utilities	09/04	[19] Lea (2004)
.Net Frame-work (v4)	Task Parallel Library (available in all .Net languages)	04/10	[20] Microsoft (2010)
Python	Stack-less Python	01/00	[21] Tismer, (2000)
C++	Intel® Threading Building Blocks	08/06	[22] Reinders (2007)

One of the primary benefits of the multicore architecture is that such systems "feature more even power density and do not show dramatic temperature peaks" [23]; nevertheless "the power consumption of the basic multicore component is critical to its cost and operation" [18].

4. Case Study: Visualization of large Neurophysiological datasets

This case study of the VISA (Visualisation of Inter Spike Associations) software demonstrates that concurrent programming is already a necessity in current applications. This software is the main output of an established research VISA project at the Visualization Lab, University of Plymouth. The main aim of this research is to develop a new approach to the analysis of experimental data in neurophysiology based on the use of modern computer science techniques such as graphical engineering, visualization and virtual reality. This new approach will provide neuroscientists with an interactive environment within which to explore their data sets. The primary focus of this research is on the analysis of multi-dimensional spike train datasets.

4.1 VISA Project Goals

Due to increasing size of data, the VISA software was completely redeveloped to fully exploit the Java concurrency framework. It was essential for the software to maintain throughput in order to achieve three main project goals:

- To enable the Neurophysiologists to process exponentially larger datasets than earlier versions in order to manage current and future demand.
- To develop the software in a language that offered cross platform compatibility to enable easy distribution
- To target a "typical researchers computer" with the option to execute with increased efficiency and speed on more powerful systems.
- To introduce a workflow based interface loosely based on "visual programming languages". This workflow interface would enable users to control the ordering of data pre-processing operations. This would provide users with greater control over the types and ordering of pre-processing operations. Ultimately the goal is to enable users to create their own pre-processing code modules to fine-tune their workflows in the future

4.2 Spike train data

In general, a neuron accumulates electrical stimulus, from other neurons coupled to it, until some internal threshold is reached. Once its threshold is reached, the neuron initiates an action potential. When a neuron initiates action potentials over time, we say that the neuron is firing. Note that action potentials are more commonly referred to as spikes and a series of these spikes over time is known as a spike train. Spike train data is one of the main types of data collected during Neurophysiological experimentation.

Spike train datasets are records of the activity of a collection of neurons under investigation. It is well established that information is encoded in this data. In the VISA project, the spiking frequency and thus, inter-spike-intervals carry information. Therefore, research is focused on the analysis of multidimensional spike train data to uncover information about the synchronisation of spike trains and the connectivity of neurons.

4.3 Quantity of Data

VISA is currently in its third edition. The first two development cycles of the VISA project incorporated the development of a cross platform tool [24, 25].

When the project began, laboratories were typically recording from at most 64 electrodes simultaneously. This was deemed to be a very large amount of data and it was recorded using an 8 by 8 multi-electrode array. Currently, due to recent improvements in electrode hardware, Neurophysiologists are now able to routinely capture data using 4096 electrodes simultaneously. For example, the Plexon Array [26], can now record data for over 30 minutes at a sampling frequency of 7.702 kHz when matched with appropriate hardware.

With typical trials lasting anywhere from a few milliseconds to tens of minutes, datasets sizes have vastly increased. For example: During an experiment when recordings are taken from all 4096 electrodes sampling data every millisecond, a 30 minute trial would result in a data file of 10.43 MB with approximately 1 million data points.

4.4 Introduction to VISA

The VISA[3] interface is effectively a visual programming language which enables the user to create a workflow.

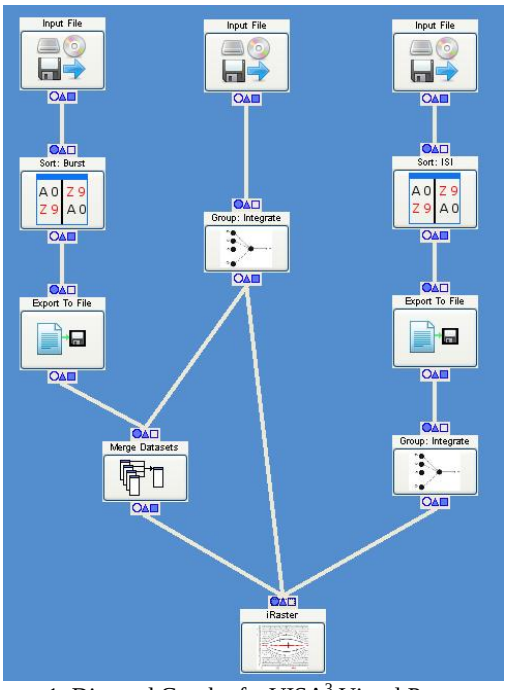


Figure 1: Directed Graph of a VISA[3] Visual Program

A workflow is a set of processes joined together. It is understood that such a programming model is naturally parallelisable; processes are able to execute as soon as all inputs are available.

A typical VISA[3] workflow is shown in Figure 1. This workflow shows 12 processes on the workflow interface. Note that these processes are selected by the user from a Toolbox and dragged on to the interface. These processes have parameters set by the user and they are connected together into the workflow as shown by dragging the mouse between process "ends". This workflow is dealing with three input files, two of which subsequently sorted and then exported to an external file. Eventually, all this data is visualised using the iRaster visualization.

The iRaster visualization enables the Neurophysiologists to investigate their data interactively using a traditionally-based raster plot representation. In addition to the functionality delivered by classical raster plot visualization, iRaster also provides the user with the following core functionality: data zooming methods, multiple views of data with view synchronization, and the contextual labelling of spike train identifiers and time points. iRaster also enables users to interactively and dynamically remove spike trains or time periods and to zoom in onto specific time periods to look at spike trains in greater detail. The software also provides an extensive list of spike train reordering functions. The majority of software tools currently available provide the usual static raster plots that are merely snapshots of the spike train data. In contrast, iRaster enables the user to interactively navigate through and directly manipulate spike train data, providing a dynamic experience of the data.

4.5 Concurrency in the VISA Interface

The key benefits of this interface are apparent. Scientists are not confronted with the need to learn a text based interface, the interface is designed to be intuitive.

The structure of the workflow, which may exploit parallel execution, is also apparent. If you refer to Figure 1, three sections of the workflow that could execute in parallel are obvious. Each process, such as the Import File process or the "Merge Datasets" process, represents a processing activity that may commence execution as soon as its inputs are available. The connections between various processes show the precise flow of data within the program.

Within the Java language, parallel execution is achieved through the concurrency framework. It operates as follows:

A 'pool' of threads each capable of executing a code module is created. Note this applies only to code modules that implement the concurrency frameworks 'Callable' interface. Pool size is dynamically determined based on the processors available on the executing machine but aims to optimise for CPU bound tasks. It is reasonable to assume that most tasks in VISA[3] will fall into this category.

At the beginning of execution the application determines all paths, through the directed graph, that comply with the requirements of the information processing cycle. Specifically this ensures that some input is received (usually from a data file), some processing is performed and finally some output is generated (typically a data visualisation module is triggered).

Each processing node of the directed graph is implemented as a "binary latch" [11]. This latch acts a synchroniser which simulates a gate that can only be opened once. Until the conditions for opening the gate are met, all threads reaching the latch will be unable to proceed. The latch will be converted into the terminal state once these conditions are met. In the case of VISA[3], the conversion requires all preceding operations to have completed and delivered a dataset to the waiting process. When the gate opens, all datasets from previous nodes are delivered to the next node and processing begins on that thread.

The datasets flowing through the directed graph are required to implement the VISA[3] interface IProcessingResult. Note that IProcessingResult is designed to wrap any data structure. Therefore, classes implementing this interface serve as the 'tokens' that drive the dataflow processing [27] and are placed onto the nodes input/output paths (triggering the 'latches' as required).

In addition to the development of a workflow interface, the Java Concurrency Framework was further leveraged to improve performance of the iRaster visualisation. As described in section 4.4, this visualisation provides an interactive raster plot of the spike trains.

The re-engineering of the visualisation was critical to maintain interactivity, due to the increase in dataset size. The original implementation typically worked with several hundred spike train recordings simultaneously whereas the new version works with thousands. Note that scientists are particularly keen to embrace this software as their traditional means of analysis do not scale.

4.5.1 Exploiting multiple-cores in VISA

Maintaining the responsiveness of an interactive application while filtering and processing this amount of data is challenging. Given this complexity, effective use of computing power in the rendering and display of the raster plot is essential. Recall that it has already been established that future increases in computer power will be delivered through multiple compute cores. The Java Concurrency Framework provides a natural means to access and manage this increased computing power in the future.

To exploit multiple cores, the VISA[3] data model was redesigned as a set of thread safe classes that could be shared across multiple executing tasks in the Concurrency Framework. The iRaster visualization process was designed using the model, view, controller design pattern with the synchronisation of shared data occurring in the model. Subsequently, the rendering could easily be adapted for parallel execution.

This required a concurrent task that accepted a screen area and spike train to render within the area. Therefore, rendering the display focuses on the individual components of the data model that are to be shown as well as generating a collection of concurrently executable tasks. A latch is again employed to ensure the various tasks complete before the final display is presented. The raster chart is then cached with re-rendering occurring only as data is filtered into/out of the display. Interactive components, such as selection highlights, scale sliders, are sequentially rendered over the cached chart.

4.6 Future VISA3 Development plans

The workflow interface and the iRaster visualisation have clearly demonstrated that parallel execution of data pre-processing and visualisation activities can scale the application to 2000+ data recordings each with thousands of data points while maintaining responsiveness. There are now two primary tasks. The first task is the conversion of the iGrid visualisation [24] to use the Java Concurrency Framework to enable the new larger datasets to be visualized in this way. This will enable the popular visualization iGrid to exploit multicore processors, thus enabling the visualization of new, larger datasets. This is challenging as iGrid is based on the production of numerous cross-correlograms. These calculations become so numerous that it is likely to be necessary to move to general-purpose computing on graphics processing units (GPGPU) model to ensure the user requirement for responsiveness are delivered.

The second task is to deploy the iRaster software more widely. The CARMEN Project is developing a 'workflow' system similar to the VISA interface [28] for processing neural recordings. In principle, this should be directly connectable to the iRaster visualisation. Whilst it requires conversion to a client-server model to support deployment on the CARMEN hardware, it will be offered as a downloadable client to view data stored in the CARMEN repository. The intention is to make this software freely available for all non-commercial use.

5. Conclusions

Multi-Threaded code originally arose from the need to manage long running tasks without the executing application becoming unresponsive to the user. However, over the last decade a new use of this technique has arisen. This is attributed to the fact that hardware manufacturers now deliver increased computing performance through multiple compute cores. Threads have been used as a natural way of writing applications that fully exploit delivery of computing power. Nevertheless, this application of threads falls outside their original purpose. Therefore, its use has been limited by a lack of software tools and developer training.

Now that hardware manufacturers have committed themselves to multi-core hardware systems, the software developer must acknowledge that to continue to benefit from ever faster hardware, the way in which they write programs must change.

Developers are beginning to accept this need to change and this is supported by the availability of development tools such as Intel® Parallel Studio suite and the addition of concurrency frameworks to major programming languages.

Despite the availability of these support tools, threading and concurrency continue to be seen as advanced concepts with training continues to lag behind the deployment of hardware capable of executing concurrent code.

In the next few years, this must change with the next generation of developers being trained to expect their code to execute in a concurrently on multi-core hardware. In many research fields multi-core hardware will be combined with clustering and grid technologies (cloud computing) to dramatically increase data processing throughput. In business, every employee can expect to be using multi-core devices as standard (from desktop PC to mobile phones and tablets). The developers of today graduate with a firm understanding of object orientated development; the developer of tomorrow will need to add concurrent programming and the experience of a concurrency framework to these skills if they are to meet the needs of business tomorrow!

The value of these skills has been demonstrated in the case study of the VISA^3 project. In VISA^3, the number and length of neural spike train recordings that need to be processed by the software has been scaled up by a factor of ten. Finally the application of a workflow based interface to the VISA^3 software demonstrates that visualising a concurrent application in this manner may offer a means to effectively design and debug concurrent software.

6. References

[1] ARIS, J., HERMON, P., LAND, F. & CAMINER, D. 1997. L.E.O.: The Incredible Story of the World's First Business Computer Mcgraw-Hill.

[2] SHNEIDERMAN, B. & PLAISANT, C. 1998. Designing the user interface: Strategies for effective human-computer interaction Addison Wesley.

[3] MICROSOFT. 1995. Windows 95 Architecture Components Windows TechNet [Online]. Available: http://technet.microsoft.com/en-us/library/cc751120.aspx [Accessed 14/03/2012].

[4] APPLE. 2001. Threading Architectures - Technical Note TN2028 [Online]. Apple Inc. Available: https://developer.apple.com/legacy/mac/library/#technotes/tn/tn2028.html#//apple_ref/doc/uid/DTS10003065 [Accessed 26/03/2012 2012].

[5] AIKAT, D., STEPNO, B., CHERNOFF, E., MANNING, M., ROBINSON, W. & HUGHES, T. 1995. The Digital Research Initiative - What is UNIX [Online]. Chapel Hill: University of North Carolina. Available:

http://www.ibiblio.org/team/intro/unix/what.html [Accessed 05/03/2012 2012].

[6] CORPORATION, Intel. 2008. What does it mean to be I/O Bound. Intel Corporation.

[7] DUBASH, M. 2005. Moore's Law is dead, says Gordon Moore [Online]. Techworld. Available: http://news.techworld.com/operating-systems/3477/moores-law-is-dead-says-gordon-moore/ [Accessed 22/09/2011].

[8] KRAUSS, L. M. & STARKMAN, G. D. 2004. Universal Limits on Computation [Online]. Available: http://arxiv.org/pdf/astro-ph/0404510v2.pdf [Accessed 13/03/2012 2012].

[9] HILL, M. D. & MARTY, M. R. 2008. Amdahl's Law in the Multicore Era. Computer - IEEE Computer Society, 33-38.

[10] MYSLEWSKI, R. 2009. The multicore future, and how to survive it - Avoiding the proprietary extensions trap [Online]. San Francisco. Available: http://www.theregister.co.uk/2009/08/25/multicore_developments/ [Accessed 05/03/2012 2012].

[11] GOETZ, B., PEIERLS, T., BLOCH, J., BOWBEER, J., HOLMES, D. & LEA, D. 2006. Java Concurrency in Practice, Pearson Education.

[12] APPLE. 2009. Grand Central Dispatch - A better way to do multicore [Online]. Apple Inc. Available: http://www.ctestlabs.org/hughes_multicore/documents/GrandCentral_TB_brief_20090608.pdf [Accessed 26/03/2012 2012].

[13] GHULOUM, A., SHARP, A., CLEMONS, N., TOIT, S. D., MALLADI, R., GANGADHAR, M., MCCOOL, M. & PABST, H. 2010. A Flexible Parallel Programming Model for Multicore and Many-Core Architectures [Online]. Intel Software and Services Group. Available: http://drdobbs.com/cpp/227300084 [Accessed 14/03/2012 2012].

[14] PATTERSON, D. A. & HENNESSY, J. L. 2008. Computer Organization and Design: The Hardware/Software Interface Morgan Kaufmann.

[15] INTEL. 2008. News Fact Sheet [Online]. Intel Corporation. Available: http://download.intel.com/pressroom/kits/events/idffall_2008/IDF_Day2_FactSheet.pdf [Accessed 13/03/2012 2012].

[16] INTEL. 2009. Intel PR Chip Shots [Online]. Intel Corporation. Available: http://www.intel.com/pressroom/chipshots/archive.htm#052609a [Accessed 13/03/2012 2012].

[17] PETURSSON, H. V. 2011. Stackless Python Applications [Online]. Python Software Foundation. Available: http://www.stackless.com/wiki/Applications [Accessed 14/03/2012 2012].

[18] BLAKE, G., DRESLINSKI, R. G. & MUDGE, T. 2009. A Survey of Multicore Processors. IEEE SIGNAL PROCESSING MAGAZINE, 26-37.

[19] LEA, D. 2004. JSR 166: Concurrency Utilities [Online]. Java Community Process Program. Available: http://jcp.org/en/jsr/detail?id=166 [Accessed 06/03/2012 2012].

[20] MICROSOFT. 2010. Parallel Programming in the .NET Framework [Online]. Microsoft Developer Network. Available: http://msdn.microsoft.com/en-us/library/dd460693(v=vs.110).aspx [Accessed 06/03/2012 2012].

[21] TISMER, C. 2000. Stackless Python 1.0 + Continuations 0.6 [Online]. Available: http://mail.python.org/pipermail/python-dev/2000-January/001835.html [Accessed 26/03/2012 2012].

[22] REINDERS, J. 2007. Threading Building Blocks Outfitting C++ for Multi-core Processor Parallelism O'Reilly Media.

[23] MONCHIERO, M., CANAL, R. & LEZ, A. G. 2008. Power/Performance/Thermal Design-Space Exploration for Multicore Architectures. IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, 19, 666-681.

[24] STUART, L., WALTER, M. & BORISYUK, R. 2005. The correlation grid: analysis of synchronous spiking in multi-dimensional spike train data and identification of feasible connection architectures. Biosystems, 79, 223-233.

[25] SOMERVILLE, J., STUART, L., SERNAGOR, E. & BORISYUK, R. 2011. iRaster: A novel information visualization tool to explore spatiotemporal patterns in multiple spike trains Journal of Neuroscience Methods, 194, 158-171.

[26] PLEXON 2006. MEA Workstation - System for recording and analyzing microelectrode arrays. In: INC, P. (ed.) Online. Dallas: Plexon Inc.

[27] DENNIS, J. & ROBINET, B. 1974. First version of a data flow procedure language. Programming Symposium. Springer Berlin / Heidelberg

[28] SMITH, L. S. 2010. CARMEN - Code Analysis, Repository & Modeling for E-Neuroscience [Online]. CARMEN Consortium. Available: http://www.carmen.org.uk/ [Accessed 29/03/2011 2011].

SKALA: Scalable Cooperative Caching Algorithm Based on Bloom Filters

Nodirjon Siddikov[1] **and Dr. Hans-Peter Bischof**[2]
[1] Enterprise IT Operation and Planning Unit, FE Coscom LLC, Tashkent, Uzbekistan
[2] Computer Science Department, Rochester Institute of Technology,
Rochester, New York, USA

Abstract - *This paper presents the design, implementation and evaluation of a novel cooperative caching algorithm, called SKALA, which is based on the bloom filter data structure. The algorithm is designed using decentralized architecture and offers highly scalable solution. It resolves the problems related to an inefficient use of the manager's resources. The scalability of the algorithm is maintained by distributing the global cache information among cooperating clients. Furthermore, the memory overhead is decreased due to a bloom filter data structure which reduces the global cache size. The correctness of the algorithm is evaluated by running experiments. The experiment results demonstrate that SKALA decreases the manager load by the factor of nine which implies that it is more scalable compared to existing algorithms. The results also show a significant reduction in the memory overhead which implies that SKALA uses manager's resources more efficiently.*

Keywords: bloom filter, global and local cache, scalability, decentralized architecture, memory hierarchy

1 Introduction

The distributed systems, such as distributed file systems (DFS) and content distribution networks (CDN), usually employ caching mechanism. The primary goal of this mechanism is to temporarily store the frequently or recently accessed data, so that future data requests can be served faster. The benefit of caching can be extended by implementing a cooperative caching mechanism in these systems. Cooperative caching is implemented using two layers of memory – local and global cache. The distributed systems based on client/server architecture usually use the local cache to provide data reusability at client side. While the local cache is dedicated to specific client, the global cache is shared among all clients. The global cache's primary goal is to help clients benefit from earlier accesses (to the same data) by other clients sharing the global cache. The content of the local and global caches are managed by special caching algorithms such as *N*-chance [4] and hint-based algorithms [5].

1.1 Research problems

The existing cooperative caching algorithms, such as *N*-chance and hint-based algorithms face several problems that prevent them from being applicable for wide variety of distributed systems. First, these algorithms do not scale when the number of clients or the local cache size is increased. The primary reason for such behavior is due to a centralized architecture implemented in these algorithms. This architecture uses single manager node which is responsible for the maintenance of the global cache. When the number of clients increases, the manager is required to handle more requests coming from clients and eventually becomes overloaded. Second, existing algorithms create a memory overhead on the manager side. As the number of clients increase, the global cache size at the manager also increases gradually which causes a memory overhead.

1.2 Proposed solution

This paper addresses the research problems by introducing a novel cooperative caching algorithm, called SKALA, which is based on a bloom filter data structure. SKALA uses decentralized architecture that relieves the manager from global cache maintenance and distributes the global cache among client nodes. Thus, clients would have their own copy of the global cache so that they avoid contacting the manager for the data. This feature reduces the load on the manager to maintain the scalability of the algorithm. Moreover, SKALA decreases memory overhead by incorporating the bloom filter into the global cache. The proposed solution is suitable for distributed systems that operate in client/server environment.

In order to evaluate the performance of the algorithm, a trace-driven software simulator has been developed. The experiment results demonstrate the effectiveness and correctness of the algorithm. According to the results, SKALA performs better compared to the existing algorithms when the number of clients or the local cache size increases. In particular, the bloom filter reduces the global cache overhead by factor of nine and maintains the algorithm's scalability over many client nodes.

1.3 Motivation

The primary motivation of this paper is to measure the scalability of the cooperative caching algorithms. Another motivation is to explore the relationship between cryptography and caching algorithms. The bloom filter data structure, for example, establishes this relationship, because it uses a cryptographic hash function to insert, lookup and delete its elements. Moreover, it was also claimed that bloom filter can be applied for caching algorithms to save memory space [2]. The bloom filters and hash functions are relatively new in the current research area, thus it is interesting to see how they help cooperative caching algorithms be scalable.

1.4 Organization of paper

The current paper is organized as follows: Section 2 describes the required background knowledge and concepts. Section 3 elaborates more into existing cooperative caching algorithms. Section 4 examines the research problem and introduces the novel caching algorithm. Section 5 describes the experiments and the metrics used to evaluate the algorithms. The limitations of this paper and the future works in this research field are discussed in Sections 6 and 7, respectively. Finally, Section 8 summarizes the paper by presenting the conclusions drawn from the experiment results.

2 Background

2.1 Cache memory

A cache memory transparently stores the data so that future data requests can be served faster [9]. It can be used to increase the performance of secondary storage devices, such as a hard disk, by saving the copies of frequently or recently accessed blocks. In a client/server environment, caching is used to reduce the request latency, service time (i.e. round trip time) and the network traffic. The data stored in the cache may either be a previously computed value or the block that has been requested before.

The cache memory is a part of the computer *memory hierarchy* ordered from the fastest memory at the top to the slowest one at the bottom. In such setting, cache filters out the data request directed toward the slower layers of the hierarchy. For example, when the client requests the data block, it first checks the cache. If the requested data is in the cache (i.e. a cache hit), it is served directly from the cache memory, which is relatively faster. Otherwise (i.e. a cache miss), the data has to be fetched from the slower memory layer such as a disk. Therefore, the performance of the system will be improved if more data is read from the cache [9]. The size of a cache depends on the requirements of the system, but it is usually kept small to maintain its efficiency.

2.2 Cooperative caching system

A typical cooperative caching system in client/server environment usually implements the local and global caches. The local cache is implemented in the client side as part of its memory hierarchy. The global cache is also considered as a layer in the hierarchy and it is positioned between a local cache and the server's storage [5]. Its content is managed by a cooperative caching algorithm using the logic of least recently used (LRU) and least frequently used (LFU) algorithms.

The architecture of cooperative caching system usually involves a manager, clients and a server. The manager implements and maintains the global cache; the server hosts the data blocks and the client sends the read requests to these blocks. The manager is responsible for the coordination of the clients' requests to the blocks. The extent of coordination is different among various cooperative caching algorithms and it is considered a major distinction between them [5]. Moreover, the clients have to cooperate with each other when accessing the blocks. Cooperation takes place when one client does not have the required block in its local cache and it probes caches of other clients for the required data.

Cooperative caching systems are usually designed using a centralized architecture. In this architecture, if client experiences local cache miss, then it contacts the manager to lookup the global cache. The manager usually knows which client's cache contains the required block through its global cache. If there is a global cache hit, the manager redirects the request to the appropriate client. Otherwise, the manager fetches the block from the server which may be a slow and an expensive process. Figure 1 illustrates an example of such centralized architecture.

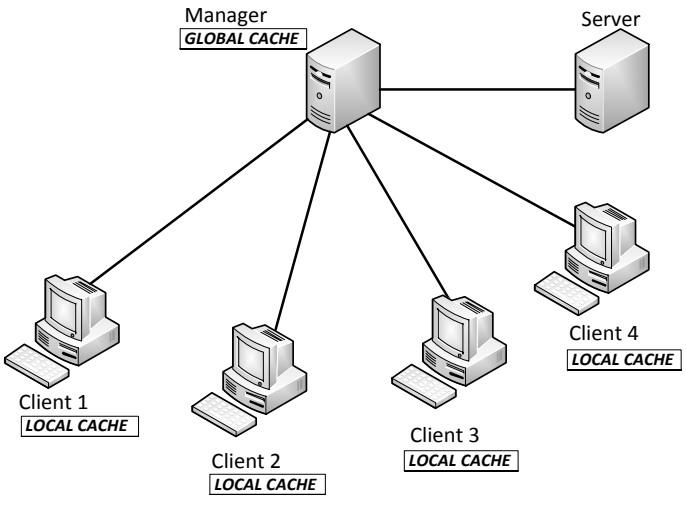


Figure 1: *A centralized cooperative caching system. In this system, a single manager is responsible to maintain the global cache.*

3 Existing algorithms

3.1 *N*-chance algorithm

The *N*-chance algorithm is one of the pioneers in the field of cooperative caching. It uses a centralized architecture which includes clients, the manager and the server. In this algorithm, the local cache is located in the clients' side. But the global cache is maintained by the manager and has the comprehensive view of the all local caches. Local cache content is managed by LRU cache replacement algorithm, but the global cache is coordinated using a more sophisticated policy. In this policy, the data block is either discarded or kept in the global cache based on its quantity in the cooperative caching system. If there is only one specific block in entire system, it is called a *singlet*; otherwise it is considered as a non-singlet. The singlet data blocks are forwarded to a randomly selected client to further retain them in the system; otherwise, they are discarded immediately. Each singlet block is given a recirculation count *N* such that every time the block is forwarded, the count is decremented. In other words, the block is given *N* chances to stay in the cooperative caching system. When the count reaches zero, the block is discarded regardless of its singlet status [4]. The central manager component plays an important role in the *N*-chance algorithm. It is responsible for global cache maintenance and lookup, and forwarding the block requests to the clients and the server. Every time client inserts or removes a block from its cache, it reports these changes to the manager. The manager uses this information about changes to maintain the consistency of the global cache [4].

3.2 Hint-based algorithm

In *N*-chance algorithm, the information contained in the global cache is considered as *facts*. This is because global cache reflects the factual data stored in the local caches of clients. However, maintenance of these factual data increases overhead on the manager. The hint-based cooperative caching algorithm attempts to solve this overhead problem by relaxing the centralized control of global cache. In particular, the algorithm distributes the portions of the global cache information among clients as *hints*, so that clients can make local caching decisions. Thus, the global cache is maintained by both clients and the manager [5]. In such setting, the manager still retains the control of the global cache, but it incurs much less overhead from clients.

Facts and hints are main components of the hint-based algorithm. They usually contain the information about the location of a *master* block, which is the original block fetched from the server. The hints reduce the client's dependence on the manager when performing caching operations. However, the hints are not always accurate, because they are local to the client and they do not reflect the changes taking place in the caches of other clients. For example, if a hint tells the client about the location of a particular master block, it is not guaranteed that the block is present at the remote client's cache. The chances are that the remote block might have been forwarded to another client or it has been discarded entirely. Also, the information about this change might not have been reflected in the client's hint. Thus, the hint gives client only a clue about the probable location of a block in the cooperative caching system. However, managing hints is less costly for a client than managing facts, because the accuracy of hints needs not to be maintained at all times [5].

4 New algorithm: SKALA

4.1 Problems with existing algorithms

There are several problems associated with existing cooperative caching algorithms. The first problem is related to their scalability – the algorithms do not scale when the number of clients increase in the cooperative caching system. Increased number of clients generate excessive amount of block requests towards the manager and causes it to get overloaded. For example, in the *N*-chance algorithm, the singlet block is relocated to the remote client through the manager. When the number of clients increases, such block relocation procedures happen more often which causes a communication overhead among the clients and the manager. In the case of hint-based algorithm, the local hints may become obsolete, thus clients will be required to contact the manager to obtain the fact. Such algorithm also produces a communication overhead as the number of clients increases. Also, it may take a while until hints are updated, so there is a risk for a client to perform an inaccurate caching operation based on the obsolete hint. The outcome of this operation is unfavorable since inaccurate decisions increase the cache miss rate [8].

Another problem of existing algorithms is associated with a memory overhead placed on the global cache. As more clients are added to the system, the manager has to allocate more memory space for the global cache. While the hint-based algorithm mitigates this problem by using hints, it still maintains facts in the global cache at the cost of manager's resource. This may result in a waste of memory resource when the number of obsolete facts becomes excessive.

4.2 Proposed solution

In order to overcome the problems with existing algorithms, current research paper proposes a solution to reorganize the components presented in Figure 1. In this solution, the global cache component is removed from the manager and distributed among the clients. Thus, each client maintains its own copy of the global cache besides its local cache. A client periodically updates the state of its global cache. Updating procedure involves calculation of bloom filter from local cache and broadcasting it among the other clients. Figure 2 shows the overall architecture of the proposed solution.

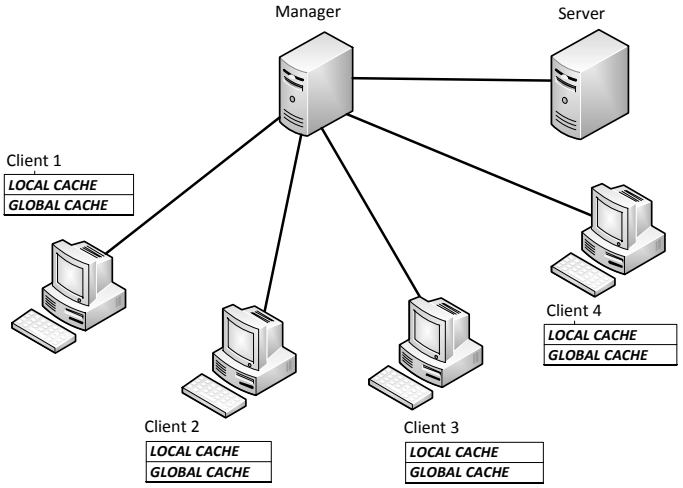


Figure 2: *The architecture of the proposed solution. In this architecture, the global cache component is removed from the manager and distributed to each client.*

The proposed solution is implemented by a novel cooperative caching algorithm called SKALA. It is designed using decentralized architecture and requires each client to maintain local and global cache components. The local cache stores the accurate state of the data block including its content. But, the global cache stores only approximate information about blocks. Due to such approximate information, the global cache lookup operation may return false positive (i.e. false cache hit) and false negative (i.e. false caches miss) results. However, in SKALA, the local cache is accessed more frequently than the global cache. Therefore, the algorithm does not keep the global cache up-to-date at all times and false positives and negatives happen occasionally within tolerable threshold. SKALA also reduces the global cache size by using bloom filter. This assures that global cache uses less space in client's memory which increases the local cache hit rate.

4.3 Data structures used in SKALA

SKALA uses traditional and novel data structures to implement the cache memory (Figure 3). The local cache is implemented using a hash table which keeps each data block as a key/value pair. The key represents the block's ID and the value denotes the block's content. Unlike a hash table, a bloom filter only stores the block's ID as a key, but the block contents are not stored [6]. Such design gives an advantage during caching operations. For example, a bloom filter spends less time to look up the block information compared to the hash table. But, the time to access the block contents stored in the local cache becomes significant because of the network latency [1].

The bloom filter data structure is considered as an exceptional feature of the SKALA. It is a probabilistic data structure which basically is an array. Similar to an array, it supports insert, delete and lookup operations. However, the result of the lookup operation is returned with some level of probability. For example, the lookup operation may result in false positives and negatives. The false negative happens when the particular block is stored in the cache, but the bloom filter says otherwise. On the other hand, the false positive occurs when the block is not stored in the client's cache, but the bloom filter says otherwise. Even though these errors affect the cache hit ratio, bloom filter keeps the logic of the cooperative caching algorithm intact. For example, a false cache hit does not result in a wrong block being served [2]. The probability of getting false positives and negatives can be controlled by proper choice of values for bloom filter parameters (such as its size and expected number of elements). One of the advantages of bloom filter is its fixed size so that the capacity of the global cache does not change over time. On the other hand, the bloom filter can only be used for cache lookup operation, and it cannot directly be used to store the block's other attributes such as its size [7].

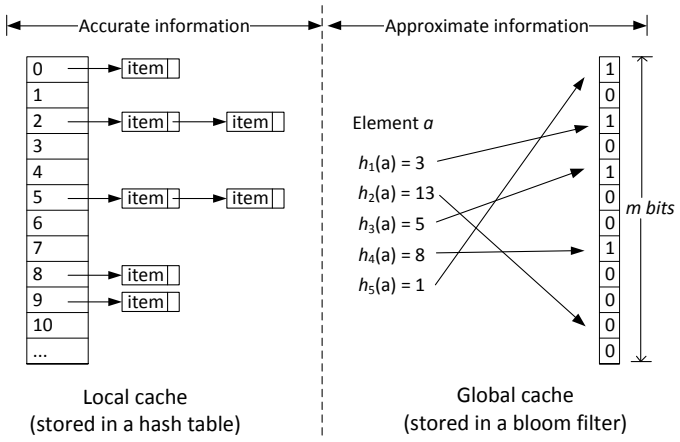


Figure 3: *The data structures used in the local and global cache.*

4.4 Algorithm limitations

Even though the SKALA has significant advantages over the existing algorithms, it exhibits some limitations. First, SKALA only works in block level granularity; thus, the algorithm requires the data to be split into blocks. Thus, SKALA accepts block ID as an input, but not a whole file name. Second, the performance of SKALA under large cooperative caching systems such as CDNs is not predictable. The algorithm is only tested with a cache memory which size is 64 – 2048 KB. Such test sizes are significantly small compared to the cache sizes of real-world distributed systems. Third, SKALA executes its operations sequentially and is not tailored to run in parallel. In order to develop a parallel version of the algorithm, a special parallel library has to be used. Also, applying parallelism into algorithm requires an external synchronization of internally used data structures.

5 Evaluation

5.1 Evaluation metrics

The following metrics are used to evaluate the performance of the cooperative caching algorithms:

- *Manager load* measures the overhead a client imposes on the manager. The overhead is expressed as a number of messages generated by the client to communicate with the manager. This metric is important because it represents the amount of work completed by the manager for cooperative caching operations.
- *Memory overhead* measures the overhead imposed on the manager's memory. The size of the global cache is used to determine the memory overhead. Such measurement is also an important factor in estimating the algorithm's scalability.

5.2 Experiment results

In order to conduct the experiments, a special software simulator is developed in Java which implements all cooperative caching algorithms under investigation. The simulator used a single Java Virtual Machine and its input parameters are adjusted through a configuration file. List of algorithms tested are provided below:

- *N*-chance algorithm;
- Hint-based algorithm;
- Non-bloom filter based algorithm;
- Bloom filter based algorithm (SKALA).

The input parameters of the simulator specific to caching algorithms (local cache size, the number of clients and the network latency) are adjusted regularly based on the requirement of the experiment.

5.2.1 Manager load

Figure 4 shows the measurement of the load imposed on the manager. The load is directly proportional to the amount of messages exchanged between clients and the manager. These messages are broken down to consistency, replacement and lookup messages. According to the figure 4, the *N*-chance algorithm imposes most load on the manager compared to the rest of the algorithms. Its centralized architecture requires clients to make frequent contacts to the manager to lookup the global cache. Another reason is because a client constantly updates the manager with the change in its local cache content for the purpose of global cache maintenance. Consistency messages put most of the load on the manager due their importance for the global cache. In the case of the hint-based algorithm, the manager load is three times less than that of *N*-chance. Despite using hints, the algorithm still requires clients to contact the manager in order to maintain the facts in the global cache. Finally, the decentralized architecture of SKALA imposes fewer loads on the manager making the algorithm scalable. Because the global cache is locally stored in each client, it does not contact the manager at all for lookup, insert and delete operations.

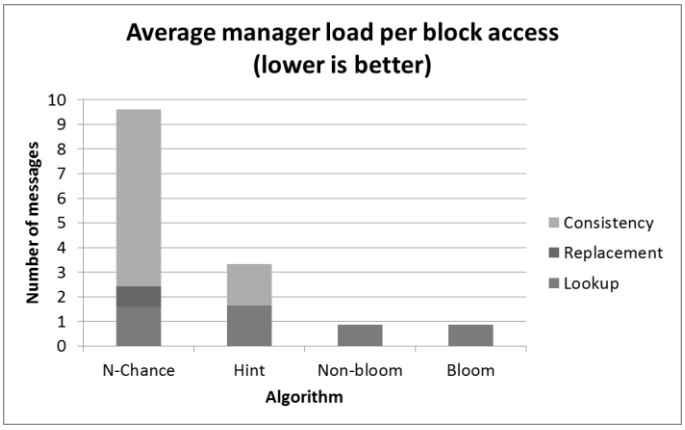


Figure 4: *The average load imposed on the manager by each algorithm. The load is defined as the number of messages sent and received by the manager. The manager load is broken down to consistency, replacement and lookup messages.*

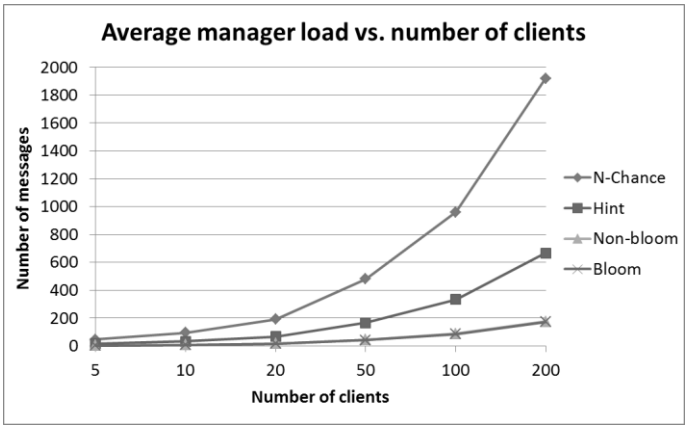


Figure 5: *The variation of the manager load as the number of clients increase.*

It is important to measure the scalability of the algorithms when number of clients increases in the cooperative caching system. If algorithm's load on the manager does not change as more clients join the system, then it is considered as a scalable. The outcome of the experiment for scalability is presented in Figure 5. According to the figure, the manager load on bloom and non-bloom filter based algorithms grows slower than the rest of the algorithms. The decentralized architecture of these algorithms keeps the communication among the clients and the manager to low regardless of the number of clients. However, the *N*-chance and hint-based algorithms demonstrate a gradual increase of the manager load under the same conditions. The reason is that algorithms require the clients to contact the manager to maintain the global cache consistency. Increased amount of messages degrade the manager's performance which eventually reduces the algorithm's scalability.

5.2.2 Memory overhead

Figure 6 shows the result of the experiment which evaluates the memory overhead of the algorithms. This research used the

value of the global cache size to estimate the memory overhead. This value is variable and depends on the aggregate sizes of local caches. According to figure 6, the memory overhead of SKALA is significantly low compared to that of other algorithms. SKALA uses a bloom filter to implement the global cache which keeps its size fixed regardless of number of blocks. On the other hand, the memory overhead is the highest in case of non-bloom filter based algorithm. Such huge difference is due to the method used to store the blocks in the global cache. This algorithm stores the exact copy of each block in the global cache. However, the rest of the algorithms use block's ID to populate the global cache and the actual block contents are stored in the local cache.

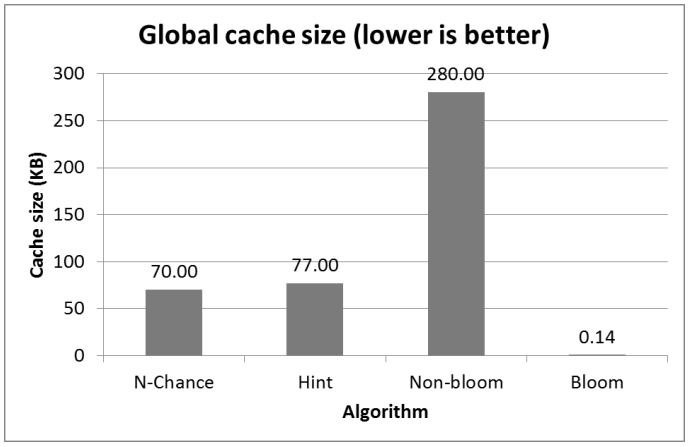


Figure 6: *The memory overhead in the cooperative caching algorithms.*

Another experiment evaluates the scalability of the algorithms when the local cache size varies between 64 – 1024 KB. The experiment assumes that the size of the global cache is proportional to size of the local caches in all algorithms. The result of experiment is shown in figure 7. In case of the non-bloom filter based algorithm, the global cache size increases exponentially as local cache size reaches the 1024 KB mark. This algorithm stores the block contents in the global cache, thus local cache content consumes more space in the global cache. *N*-chance algorithm shows similar behavior but at lesser extent due to allocation of global cache maintenance to the manager. The hint-based algorithm achieves the better result than *N*-chance through keeping merely master block information in the global cache. Finally, SKALA demonstrates immunity to the increase in the local cache size due to the fixed size of the bloom filter used in its global cache.

The final experiment uses the number of clients to evaluate the scalability of the algorithms (Figure 8). According to the figure 8, the number of clients in the system directly affects the memory overhead. *N*-chance and hint-based algorithms demonstrate that the overhead on the memory grows exponentially when number of clients increase. Such exponential growth indicates that these algorithms are not scalable due to increased block access time and increased load on the manager. Moreover, both of these algorithms use a variable size global cache which can grow infinitely. Thus, when the number of clients increases, the amount of cached block information consumes more space in the global cache. On the other hand, SKALA's global cache size remains relatively stable until the number of clients reaches 50. The global cache allocates one bloom filter for each client's local cache information, thus exceeding this threshold of 50 clients is sufficient for to increase the size of the global cache.

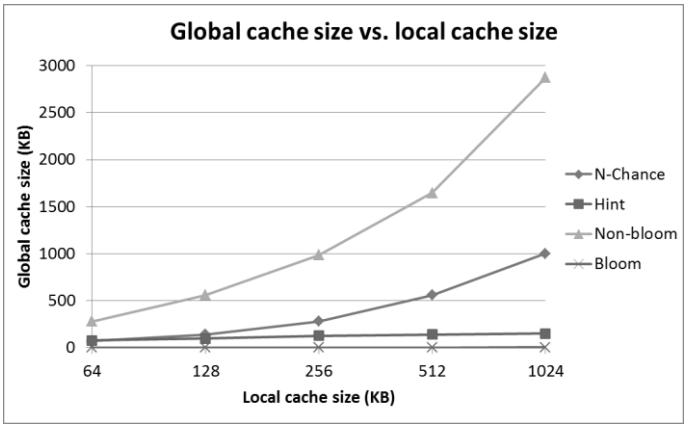


Figure 7: *The sensitivity of the memory overhead to the variation of the local cache size.*

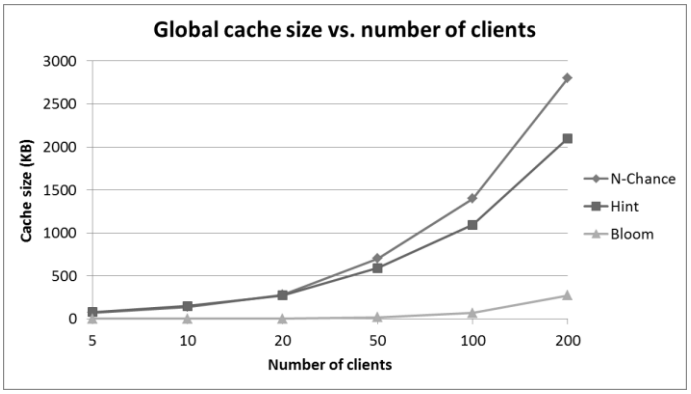


Figure 8: *The sensitivity of the memory overhead to the change in the number of clients.*

6 Limitations of proposed solution

The proposed solution faces some limitations that could affect its performance. For example, the bloom filter data structure only stores the approximate state of the global cache. This may result in false positives and negatives which means not all global cache lookup operations return accurate answer. Inaccurate answers are primary reasons for the local and global cache misses. Consequently, this increases the block access time and reduces the performance of SKALA. Therefore, the bloom filter has to be created with appropriate input parameters. Moreover, SKALA does not implement the cache

writing policy. However, integrating this policy into SKALA would make the contents of cache memory and server disk more consistent. On the other hand, such modification may not change the outcomes of the experiments, because cache writing and block reading policies are mutually exclusive and they do not affect the cache hit/miss rates. Another shortcoming of SKALA is associated with the input data used in the experiments. The input data (trace) is synthetic meaning that it is randomly generated which does not reflect the real world block access pattern. However, using real world traces would increase the credibility of the experiment results.

7 Future work

Even though SKALA proved itself as scalable caching algorithm, there are some enhancements that could be applied for the algorithm. One of them entails implementing a cache writing policy in SKALA. According to this policy, when the content of the cache is changed, the updated blocks are written back to the server's disk. Thus, the consistency of the cache memory and the disk is maintained. Another extension to SKALA is to implement an offline caching algorithm. All of the algorithms presented in this paper are all considered online algorithms. The offline algorithm can be used in the experiments to measure the upper performance bound of these online algorithms. Such measurement would help optimize the performance of the existing algorithms. Finally, implementing a multithreading would improve the performance of SKALA. For example, the computation of the bloom filter can be parallelized with multithreading so that each hash functions are calculated by separate thread.

8 Conclusion

A cooperative caching system usually implements two layers of cache memory: local and global caches. The system components usually consist of the manager, the clients and the server all of which works under the rule of the specific caching algorithm. The algorithm is also used to coordinate the contents of the local and global caches. The level of coordination is an important factor and is the main difference between various cooperative caching algorithms. The current paper introduces the bloom filter based decentralized caching algorithm, called SKALA, which focuses on the scalability and efficient use of global cache memory. SKALA achieves the scalability by removing the maintenance of the global cache from the manager. Instead, it distributes the entire global cache among the clients. The content of the global cache is composed of a set of bloom filters, one filter to represent each client's local cache. The bloom filter simplifies the implementation of the global cache replacement policy in SKALA which does not involve the manager. The experiments on SKALA show that the load on the manager is decreased due to algorithm's decentralized architecture. This assures the scalability of SKALA without reducing its performance. Moreover, the bloom filter reduces the memory overhead for the global cache. This signifies that SKALA is better at using the manager's resources efficiently compared existing solutions.

9 References

[1] Mursalin Akon, Towhidul Islam, Xuemin Shen and Ajit Singh (2010), "SPACE: A lightweight collaborative caching for clusters", Peer-to-Peer Networking and Applications, Vol.3, Iss.2, pp.83 – 99

[2] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder (2000), "Summary cache: a scalable wide-area web cache sharing protocol." IEEE/ACM Trans. Netw. 8, 3 (June 2000), 281-293

[3] Madhukar R. Korupolu and Michael Dahlin (2002), "Coordinated Placement and Replacement for Large-Scale Distributed Caches." IEEE Trans. on Knowl. and Data Eng. 14, 6 (November 2002), 1317-1329

[4] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson (1994). "Cooperative caching: using remote client memory to improve file system performance." In Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation (OSDI '94). USENIX Association, Berkeley, CA, USA.

[5] Prasenjit Sarkar and John H. Hartman (2000), "Hint-based cooperative caching." ACM Trans. Comput. Syst. 18, 4 (November 2000), 387-419

[6] Francisco Matias Cuenca-Acuna and Thu D. Nguyen (2001), "Cooperative Caching Middleware for Cluster-Based Servers." In Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC '01). IEEE Computer Society, Washington, DC, USA

[7] Purushottam Kulkarni, Prashant Shenoy, and Weibo Gong (2003), "Scalable techniques for memory-efficient CDN simulations." In Proceedings of the 12th international conference on World Wide Web (WWW '03). ACM, New York, NY, USA

[8] Woo Hyun Ahn, Sang Ho Park, Daeyeon Park (2000), "Efficient cooperative caching for file systems in cluster-based Web servers" Cluster Computing, Proceedings. IEEE International Conference on , vol., no., pp.326-334

[9] Buyya, Rajkumar et al; (2008), "Caching Techniques on CDN Simulated Frameworks," Lecture Notes in Electrical Engineering, Springer Berlin Heidelberg, URL: http://dx.doi.org/10.1007/978-3-540-77887-5_5, Accessed: October 8, 2011

[10] Dimitar, Popov et al. (2003), Cache memory implementation and design techniques, Web address: http://www.faculty.iu-bremen.de/birk/lectures/ PC101-2003/07cache/cache%20memory.htm, Accessed: October 22, 2011

Evaluating Utilization of the I/O System on Computer Clusters

Sandra Méndez[1], **Dolores Rexachs**[1], **and Emilio Luque**[1]
[1] Computer Architecture and Operating Systems Department (CAOS)
Universitat Autònoma de Barcelona, Barcelona, Spain

Abstract— *The increase in computational power of processing units and the complexity of scientific applications which use high performance computing require more efficient Input/Output (I/O) systems. In order to efficiently use the I/O system it is necessary to know its performance capacity to determine if it fulfills applications I/O requirements. Evaluating the performance capacity of the I/O system is difficult due to the heterogeneity of its architecture and the complexity of I/O software stack. It is necessary to understand the application's access patterns, the architecture, the software stack and its interaction to evaluate the I/O system utilization.*

We propose a methodology to evaluate the I/O system utilization taking into account parallel applications behavior and the I/O system performance capacity. To analyze the I/O system we evaluate the I/O path of computer clusters. We use a hierarchical scheme of the I/O system with its performance capacity at I/O library level and at I/O device level.

We propose a method to identify I/O phases of parallel scientific applications depending on its temporal and spatial patterns. We have evaluated the I/O system utilization taking into account the I/O phases behavior of applications in I/O devices.

Keywords: Parallel I/O System, I/O Architecture, Mass Storage, I/O Configuration

1. Introduction

Due to the historical "gap" between the computing and Input/Output (I/O) performance, in many cases, the I/O system becomes the bottleneck of parallel systems. In order to hide this "gap", the I/O factors with the biggest effect on performance must be identified. Furthermore, the increased computational power of processing units and the complexity of scientific applications which use high performance computing require more efficient Input/Output Systems. The parallel I/O system is complex because it is composed of different components of hardware and several layers of software. Also these I/O components are heterogeneous in computer clusters. The designer or system administrator has the difficulty either to select components of the I/O subsystem (JBOD (Just a Bunch Of Disks), RAID (Redundant Array of Inexpensive Disks) level, filesystem, interconnection network, among other factors) or to choose from different connection models (DAS, SAN, NAS, NASD) with different parameters to configure (redundancy level, stripe, among others). Programmers can modify their programs to efficiently manage I/O operations, but they need to know the I/O system, especially the I/O software stack. The system administrators and programmers need information to answer the following questions, When is it convenient to use a parallel or distributed file system? When is it convenient to use I/O nodes for management the Input/Output? When is it convenient to use RAID or single disks? When is it convenient to use local storage or remote storage? We propose a methodology to evaluate the I/O system utilization taking into account the parallel I/O of the application and the I/O subsystem. To analyze the I/O subsystem we try to cover the I/O path of data on the subsystem of computer cluster. We use a hierarchical scheme of the subsystem with its performance capacity at I/O library level and at I/O devices level. Our methodology requires extracting access patterns of the application to identify the I/O phases from to which we define an I/O model of application. To do this, we have used a tool (PAS2P [1] library) developed by our research group. This tool traces parallel applications I/O operations at MPI-IO level.

We evaluate the utilization of two I/O configurations (where we have used parallel filesystem PVFS2 and network filesystem NFS). Furthermore, we show the I/O model for two I/O benchmark of scientific applications.

The rest of this article is organized as follows: in Section II we review the related work, Section III introduces our proposed methodology. In Section IV we review the experimental validation. Finally, we present our conclusions and future work.

2. Related Work

There are several works which present evaluation of the I/O configurations for improving the performance of the I/O subsystems. Since the I/O systems are complex to evaluate, generally, the studies are focused on specific supercomputers or computer cluster.

The I/O performance analysis developed in the Sandia National Laboratories over the Red Storm platform is presented in [2]. In order to arrive at a theoretical estimation for the Red Storm configuration, they started with a single end to end path definition, across which I/O operation travels. This study differs from our work since we evaluate the I/O path at I/O Library level and at devices level taking into account the different I/O configurations of the computer

cluster. Furthermore, we use these information to evaluate how much of the I/O system capacity is being used by the application.

Kunkul and Ludwig [3] presented an evaluation of performance for computer cluster on parallel filesystem PVFS2. They introduced a systematic approach for performance analysis for parallel filesystem's architecture for various request types. This study differs from our work since we have used the IOR benchmark at I/O library level to evaluate different patterns and the IOzone benchmark at level devices to obtain the peak values of the I/O architecture.

Carns et. al. [4] presented a multilevel application I/O study and a methodology for system wide, continuous, scalable I/O characterization that combines storage device instrumentation, static filesystem analysis, and a new mechanism for capturing detailed application-level behavior. Our work differs from his study because we evaluate the I/O system utilization taking into account I/O phases of application which are used to represent an I/O model of application.

Byna et. al. [5] defined the access patterns to identify the I/O signatures for parallel I/O prefetching. They presented a classification of I/O patterns for parallel application. We use her I/O pattern classification. Our work differs from his study because we define I/O phases of the application at I/O library level. The I/O model is independent from architecture allowing us to analyze the application behavior on different I/O system configurations.

3. Proposed Methodology

In order to evaluate the I/O system configuration utilization by scientific applications it is necessary to know its performance capacity and I/O patterns of the application. We characterize the behavior of the application at I/O library level and we also evaluate its behavior on I/O configuration at I/O devices level. Figure 1 shows our methodology. This is composed of three stages: Characterization, I/O analysis and Evaluation.

3.1 Characterization

The characterization of application is done with PAS2P tracing tool. Also, we define the I/O phases of application and show the I/O model for the application. We characterize the I/O subsystem of a computer cluster following the structure of the I/O system of Figure 1. The data travels along an I/O path from application to I/O devices in writing operations and from I/O devices to application for reading operations. Depending of I/O system, data can cross over interconnection networks to arrive to data storage.

3.1.1 Scientific Application

I/O patterns of parallel applications can be divided in local patterns and global patterns. Local patterns are determined per process, showing how a file is accessed by a process. Global patterns show how the file is accessed temporally

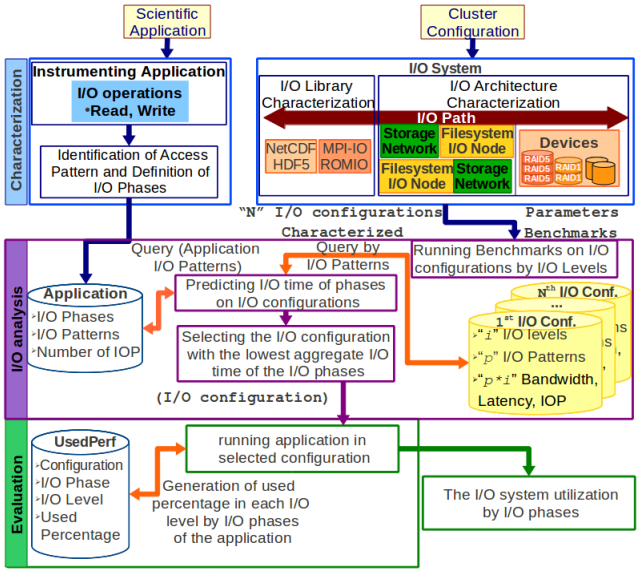


Fig. 1: Methodology to evaluate the I/O system utilization by parallel scientific applications

and spatially for processes of parallel application. We use the classification of local patterns of [5] and we define global patterns from analyzing local patterns. We define a I/O phase as a repetitive sequence of I/O patterns of a parallel application.

We represent the I/O model of an application by three major characteristics: i) the meta-data; ii) the temporal global I/O pattern; and iii) the spatial global I/O pattern. We characterize the application off-line and once at I/O library level because it provides us two important benefits. First, to obtain a model of the application's I/O independent from the execution environment, i.e. the computer cluster. Second, to evaluate the behavior of the application with different I/O configurations, avoiding the overhead of the tracing tool.

We have implemented an extension of PAS2P library to characterize application at process level. PAS2P identifies and extracts phases of the application, and by similarity analysis, this selects the significant phases (by analyzing compute and communication) and their weights. The representative phases are used to create a Parallel Application Signature which allows us to predict the application performance in target machines. PAS2P instruments and executes applications in a parallel machine, and produces a trace log. The data collected is used to characterize computational and communication behavior. The extension of PAS2P library traces I/O operations of the MPI-2 standard. MPI-2 has three aspects to data access: positioning (explicit offset or implicit file pointer), synchronism (blocking, non-blocking and split collective), and coordination (non-collective or collective). Also there are two types of file pointers (individual and shared) [6].

From MPI-IO routines we extract: type of I/O operation, initial offset, displacement, request size, filename, and file

type . Then, the traces are analyzed and we extract the access patterns for the parallel application. We incorporate the I/O primitives to the PAS2P tracing tool to capture the relationship between the computations and the I/O operations. Thus, we created a library $libpas2p-io.so$ which is loaded when the application is executed with $LD\_PRELOAD$.

We identify the I/O phases with an access pattern and their weights (number of repetitions of the pattern). With the characterization, we try to find the application phases which are used as units for behavior analysis in the I/O system. Due to the fact that scientific applications show a repetitive behavior, m phases will exist in the application. We represent a local access pattern by the tuple: (*Operation*, *initialOffset*, *displacement*, *requestSize*, *repetitions*).

The I/O phases are a repetitive sequence of same pattern on a file for a number of processes of the parallel application, thus an I/O phase is represented by the tuple: (*IDProcess[i]*, *accessPattern[i]*, *weight*, *accessMode*, *accessType*, *FileName*) where $i \in \{0..Pph\}$, Pph is the number of processes of the I/O phase.

To explain the I/O phases' identification we present an example for the benchmark MADBench2 [7]. MADbench2 is a tool for testing the overall integrated performance of the I/O, communication and calculation subsystems of massively parallel architectures under the stress of a real scientific application. MADbench2 is based on the MADspec code, which calculates the maximum likelihood angular power spectrum of the Cosmic Microwave Background radiation from a noisy pixelated map of the sky and its pixel-pixel noise correlation matrix. MADbench2 can be run as single or multi-gang; in the former all the matrix operations are carried out distributed over all processors, whereas in the latter the matrices are built, summed and inverted over all the processors (S & D), but then redistributed over subsets of processors (gangs) for their subsequent manipulations (W & C).

MADbench2 can be run on IO mode, in which all calculations/communications are replaced with busy-work, and the D function is skipped entirely. The function S writes, W reads and writes, C reads. This is denoted as S_w, W_w, W_r, C_r. MADbench2 reports the mean, minimum and maximum time spent by each function during calculation/communication, busy-work, reading and writing in each function. Running MADbench2 requires a n^2 number of processors.

By tracing MADBench2 with our tool we have obtained its metadata:

- Individual file pointers, Non-collective I/O operations, Blocking I/O operations.
- Sequential access mode, Shared access type.
- A file shared by all processes

Figure 2 shows the first phase which is composed of eight writing operations, in the second phase all processes read their positions two first, the third phase is composed of six

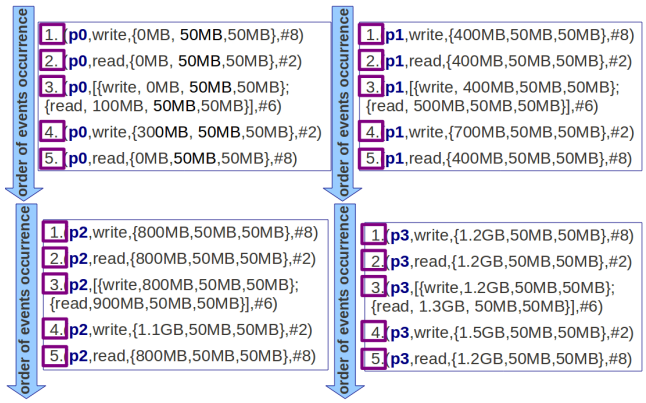


Fig. 2: Temporal and Spatial I/O pattern of MADBench2 for 4 processes (process, operation, {initialOffset, displacement, requestSize}, #repetitions)

writing/reading operations, in the fourth phase, processes write their last two positions, and in the fifth phase, processes read all their positions. This information is useful to determine the spatial pattern (the view of file and the access mode by process). However, to define the I/O phases we also need the temporal pattern (order occurrence of I/O operations on the file). Figure 2 shows the global temporal pattern of MADBench2. Note that each process shows a similar pattern (the events occur in the same order). If we analyze the phase 3 of process 0 it observes that a process repeats a writing operation six times (#6) on initial offset 0 and displacement 50 MB and request size 50MB, this is followed by a reading operation on initial offset 100MB and displacement 50MB with a request size 50MB. Figure 2 shows this pattern in the 4 processes, but with different initial offsets because each process works on disjoint sets of contiguous data.

We define five I/O phases for MADBench2 due to the similarity of its I/O patterns. Figure 2 shows the five I/O phases in purple boxes.

3.1.2 I/O System

In this step we identify the I/O system configuration. An I/O configuration depends on number and type of filesystem (local, distributed and parallel), number and type of network (dedicated use and shared with the computing), state and placement of buffer/cache, number of I/O devices, I/O devices organization (RAID level, JBOD), and number and placement of I/O node. For example, we define the following I/O configurations for an own cluster which is named Aohyper. This has the following technical characteristics: 8 nodes AMD Athlon(tm) 64 X2 Dual Core Processor 3800+, 2GB RAM memory, 150GB local disk. Local filesystem is Linux ext4 and two global filesystem: NFS and PVFS2. The NFS server has a RAID 1 (2 disks) with 230GB capacity and RAID 5 (5 disks) with stripe=256KB and 917GB capacity, both with write-cache enabled (write back). The nodes of

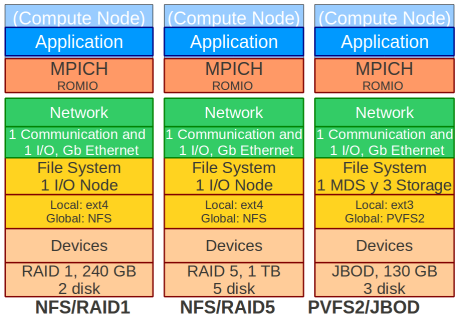


Fig. 3: I/O system configurations for Cluster Aohyper

PVFS2 are Intel(R) Pentium(R) 4 CPU 3.00GHz processors, 80 GB disk, and a Gigabit Ethernet network interconnection for data and communication.

The local filesystem is ext4 Linux for computing nodes and server NFS, and ext3 for nodes PVFS2. NFS server is an I/O node for shared accesses and there are eight I/O nodes for local accesses where users are responsible for data sharing. There are three storage nodes and a metadata Server for PVFS2 filesystem. PVFS2 is used by applications that require parallel accesses to files. The I/O nodes and computing node are interconnected with two Gigabit Ethernet networks, one for communication and the other for data. Figure 3 shows I/O configurations used in the cluster Aohyper.

3.2 Input/Output Analysis

There are three requirements that an I/O system must provide: storage capacity, availability and performance. Our work is focused on performance capacity. The performance is expressed by transfer rate (MB/sec) metrics, IOPs (Input/Output operations per second) and latency. This performance capacity is different at each I/O system level. The performance also depends on the connection of the I/O node, the management of I/O devices, placement of I/O node in network topology, buffer/cache state and placement, availability data and service. We obtain the performance capacity of I/O system at I/O library level for different access patterns through benchmark IOR [8]. Furthermore, we obtain the peak value at I/O devices level with benchmark IOzone [9].

We have executed the benchmarks in each I/O configurations with different I/O patterns and we have generated a data base by configuration with performance measures (bandwidth, latency, iops). The following data structure is used to store the I/O system performance (I/O devices, local and global filesystem, and I/O library): Operation Type (enumerate {write,read}), Request size (long integer (MBytes)), Access Type (enumerate {0 (Unique), 1 (Shared)}), Accesses Mode (enumerate {0 (Sequential), 1 (Strided), 2 (Random)}), Transfer Rate (double (MBytes/second)), Latency (double (microsecond)), Number of processes (integer).

To compare the I/O pattern of the application with benchmarks, we define the following data structure for I/O phases: ID of phase (integer), ID of files (integer), Operation Type (enumerate {write,read}), Request size (long integer (MBytes)), Access Type (enumerate {0 (Unique), 1 (Shared)}), Accesses Mode (enumerate {0 (Sequential), 1 (Strided), 2 (Random)}), Number of repetitions (integer), Number of processes (integer).

We analyze the I/O phases of the application and its weight to select the candidate configurations. We search the I/O patterns of phases on performance databases. Then, we calculate the I/O time for the I/O phases and we select the I/O configurations with the lowest I/O time. We applied the following algorithm to calculate the I/O time:

- Reading of patterns from each I/O phase
- Searching on the file of Performance the characterized transfer rate in the different I/O path levels based on I/O phases of application.
- The I/O time of the application is calculated with $\frac{DataTransferred(phase[i])}{transferRate}$ by each I/O phase, where i is the number of phase, *DataTransferred* is the data amount read or written in the phase i, *TransferRate* is the characterized value at I/O library level.

The algorithm to search the transfer rate for the I/O pattern is explained with the following steps:

- Opening the table of performance and setting the variable "found" to stop the searching when the values are found.
- If the operation type, access mode, and access type are equal to a value in the performance table, and the block size of the operation is:
  - less than minimum block size of the performance table then it selects the transfer rate corresponding to minimum block size.
  - greater than maximum block size of performance table then, it selects the transfer rate corresponding to the maximum block size.
  - equal to a block size of the performance table then it selects the transfer rate corresponding to such block size.
  - a value between the characterized values then it selects the closest upper value to the searched value.
- When the search finishes then the performance table is closed and the *TransferRate* is returned.

To explain this stage we evaluate the I/O phases to MADBench2 for 16 processes, 8KPIX, shared filetype, 32 MB request size. Figure 4 shows the I/O phases of MADBench2. Table 1 shows the I/O phases of MADBench in a general format. The I/O model of MadBench2 is represented with Fig. 4, Table 1 and its metadata.

We can observe that MADBench2 always writes and reads at the same I/O phases, when the number of processes increases the problem is divided among processes, therefore each process writes or reads a smaller block size. When

Table 1: I/O phases description of MADBench2 for 16 processes

Phase	#Oper.	InitOffset	Rep.
1	16 write	$IdProcess * 8 * 32MB$	8
2	16 read	$IdProcess * 8 * 32MB$	2
3	16 write 16 read	$IdProcess * 8 * 32MB$ $IdProcess * 8 * 32MB + +2 * 32MB$	6 6
4	16 write	$IdProcess * 8 * 32MB - -2 * 32MB$	2
5	16 read	$IdProcess * 8 * 32MB$	8

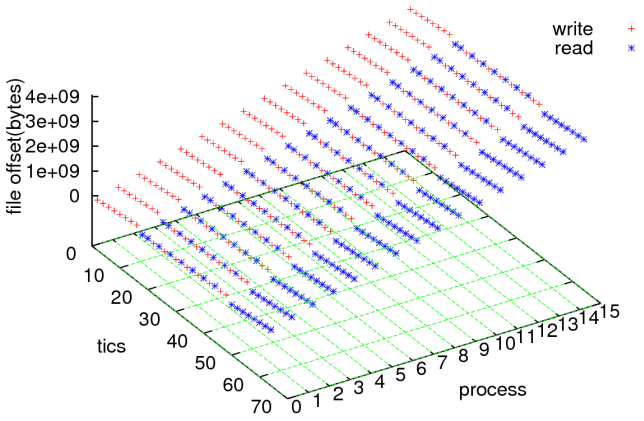


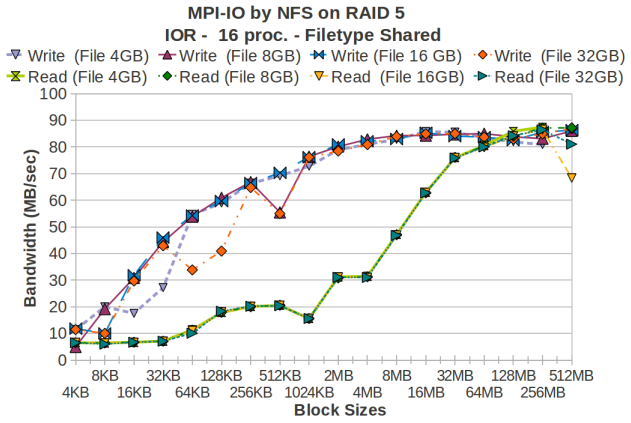
Fig. 4: I/O phases of MADBench2 where "tics" indicates the order of events occurrence

problem size increases (increases KPIX) then block size also increases.

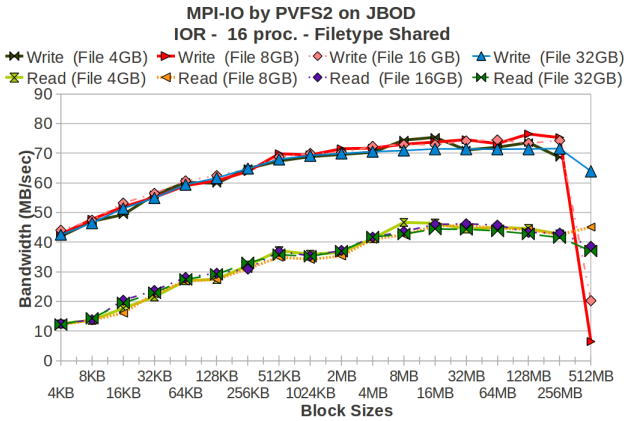
Following our example, we show in Figure 5 the characterization at I/O library level for I/O configurations of cluster Aohyper. We have selected NFS on RAID5 and PVFS2 on JBOD configurations to evaluate their utilization by MADBench2.

The results for reading operations shown in Figure 5(a) present lower performance values in NFS on RAID5 when working with block sizes below 8MB. In contrast, the transfer rate of writing operations increases proportionally to the increase in the block size. As could be seen in Figure 5(a), both writing and reading operations behave similarly for block sizes over 64MB. It is possible to infer from Figure 5(b) that in PVFS2, the transfer rate of reading operations is lower than the rate of writing operations. These results show that: the block size is the I/O factor with the highest impact in performance; while the evaluated file sizes have no high impacts in performance. Additionally, results of Figure 5(b) show a drop in performance for block size of 512MB for 16 processes, caused by timeouts in the file system due to network congestion.

The results of the characterization of both file systems, NFS (Figure 5(a)) and PVFS2 (Figure 5(b)), show two important things: i) PVFS2 on JBOD presents bigger transfer rates when working with smaller block sizes ($< 64KB$); and ii) NFS on RAID5 achieves greater transfer rates for bigger block sizes ($> 64MB$).



(a) I/O library on NFS



(b) I/O library on PVFS2

Fig. 5: I/O library on global filesystem Characterization

3.3 Evaluation

We evaluate the utilization of I/O system by the relation between the bandwidth characterized (BW-CH) and measured (BW-MD) where BW-CH depends on access patterns of each I/O phase. We define the utilization of I/O system as: $SystemUsage = \frac{BW-MD}{BW-CH}$. When a phase has two or more I/O operations then the BW-CH is calculated how the average of the BW-CH of each I/O operation that composes the I/O phase.

We can observe in the stage of I/O analysis that an application with a block size of 32MB, sequential access mode, and shared file will obtain higher transfer rates in NFS on RAID5 than in PVFS2 on JBOD (Figure 5). With this information we can select an candidate I/O configuration where possibly the application will have more performance. However, it is worth noting that we are not analyzing the absolute I/O time, because NFS on RAID5 and PVFS2 on JBOD are not directly comparable in terms of the storage capacity and the speed of devices. We compare performance results of these two I/O configurations from the characterization introduced in subsection 3.2 to observe how much from of the I/O capacity is used in each I/O configuration.

Table 2: I/O system utilization, BW-CH and BW-MD in MB/second for MADBench2 with 16 processes, file size 4 GB and a shared file on NFS

Phase	#Oper.	Data transferred	BW-CH on NFS	BW-MD on NFS	System Usage
1	128 W	4096 MB	85	93	1.09
2	32 R	1024 MB	76	68	0.90
3	192 W-R	6144 MB	80	63	0.79
4	32 W	1024 MB	85	89	1.04
5	128 R	4096 MB	76	66	0.84

Table 3: I/O system utilization, BW-CH and BW-MD in MB/second for MADBench2 with 16 processes, file size 4 GB and a shared file on PVFS2

Phase	#Oper.	Data transf.	BW-CH on PVFS2	BW-MD on PVFS2	System Usage
1	128 W	4096 MB	71	60	0.84
2	32 R	1024 MB	44	39	0.87
3	192 W-R	6144 MB	58	34	0.59
4	32 W	1024 MB	71	54	0.75
5	128 R	4096 MB	44	37	0.84

Table 2 shows the utilization of NFS on RAID5 and Table 3 shows the utilization of PVFS2 on JBOD. We also show the amount of data transferred in each I/O phase, the number and type of I/O operation (W=write, R=read, W-R=write-read), BW-CH and BW-MD in MB/second.

We can observe in Table 2 that the I/O system for the NFS on RAID5 is used at about 100% in phases with writing operations (phases 1 and 4) and 85% in phases with the reading operations (phases 2 and 5). This influences in the I/O time of phases 1 (49 sec.) and 4 (11 sec.) which are lower than in phases 5 (61 sec.) and 2 (16 sec.) in spite of to transfer the same amount of data.

Table 3 shows that the I/O system for PVFS2 on JBOD is used at about 80% in phases with writing operations and 85% in phases with the reading operations. I/O time in phases 2 (17 seconds) and 5 (43 seconds) is lower than in phases 4 (18 seconds) and 1 (61 seconds).

We can conclude from the evaluation that PVFS2 on JBOD is more efficient in the system usage in reading operations and NFS is more efficient in the system usage in writing operations. Finally, we can say that NFS on RAID5 provides lower I/O time than PVFS2 on JBOD to MADBench2.

4. Experimentation

To validate the methodology, the I/O phases identification is applied to Block Tridiagonal(BT) application of NAS Parallel Benchmark suite (NPB)[10]. The BTIO benchmark performs large collective MPI-IO writes and reads of a nested strided datatype, and it is an important test of the performance a system can provide for non-contiguous workloads. After every five time steps the entire solution field, consisting of five double-precision words per mesh point, must be written to one or more files. After all time steps

Table 4: I/O phases description of NAS BT-IO subtype FULL for 16 processes

Phase	Oper.	InitOffset	Rep.
1-40	16 W in each phase	$RS * IDProcess+$ $+(RS*(N-1)+$ $+(RS*15)*(N-1))$	1
41	16 R	$RS * IDProcess+$ $+(RS*(Rep-1)+$ $+(RS*15)*(Rep-1))$	40

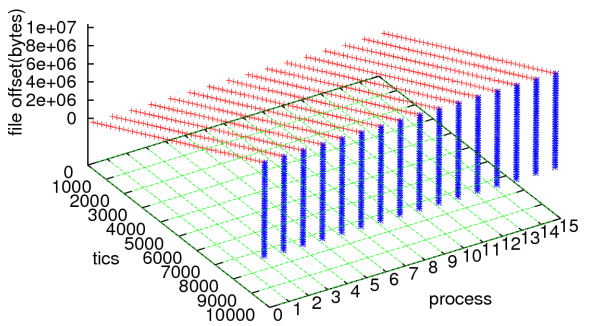


Fig. 6: I/O phases of NAS BT-IO

are finished, all data belonging to a single time step must be stored in the same file, and must be sorted by vector component, x-coordinate, y-coordinate, and z-coordinate, respectively. We have obtained the following meta-data of NAS BT-IO in the FULL subtype with our tool:

- Explicit offset, Blocking I/O operations, Collective operations.
- Strided access mode, Shared access type.
- MPI-IO routine MPI_Set_view with etype of 40.
- Request size 10MB.

Figure 6 shows I/O phases for 16 processes, Class C and FULL subtype. Table 4 shows description of I/O phases for NAS BT-IO where N is the number of phase $(1-40)$ and RS is request size, $IDProcess$ is the rank of MPI processes $(0-15)$.

We have executed NAS BT-IO in the Aohyper cluster: NFS on RAID5 and PVFS2 on JBOD, and we have evaluated their utilization.

4.1 Evaluation

Table 5 and Table 6 show the utilization of the I/O system on NFS and PVFS2. We also show the amount of data transferred in each I/O phase, the number and type of I/O operation (W=write, R=read, W-R=write-read), BW-CH and BW-MD in MB/second.

Table 5: Bandwidth Characterized (BW-CH) and Measured (BW-MD) in MB/sec for NAS BT-IO with 16 processes, file size 6.4GB and a shared file on NFS

Phase	#Oper.	Data transferred	BW-CH on NFS	BW-MD on NFS	System Usage
1 to 40	640 W	6.4GB	85	65	0.76
41	640 R	6.4GB	63	47	0.74

Table 6: Bandwidth Characterized (BW-CH) and Measured (BW-MD) in MB/sec for NAS BT-IO with 16 processes, file size 6.4GB and a shared file on PVFS2

Phase	#Oper.	Data transferred	BW-CH on PVFS2	BW-MD on PVFS2	System Usage
1 to 40	640 W	6.4GB	71	58	0.81
41	640 R	6.4GB	44	45	1.02

We observe that the I/O patterns of NAS BT-IO will obtain similar performance in both configurations, as infered from the I/O analysis stage presented in subsection 3.2.

NAS BT-IO reported that NFS on RAID5 has lower I/O time than PVFS2 on JBOD with a difference of 14 sec. (NFS on RAID5=244 seconds and PVFS2 on JBOD=259 seconds). PVFS2 on JBOD uses 80% of I/O capacity for writing operations, and 100% to reading operations; while NFS on RAID5 uses 76% of its I/O capacity for writing operations, and 74% in reading operations.

In principle, the two I/O configurations are suitable to I/O phases of NAS BT-IO FULL in its class C. When the problem is increased, NFS on RAID5 presents some interesting characteristics because of its I/O devices (higher storage capacity and data availability). However, PVFS2 on JBOD can be more suitable in situations where there are multiple processes accessing to a shared file (dynamic distribution of data through its scheme of the metadata and I/O servers), despite having less capacity and speed than NFS on RAID5. Besides, it is an option with low cost because we have configured PVFS2 with single disks and we have obtained acceptable performance.

5. Conclusion

A methodology to evaluate the I/O system utilization of parallel computers for the parallel scientific applications has been proposed and applied. It allows us to determine how much of the system's capacity is being used. The methodology can be used for selecting the more convenient configuration from the available configurations in an I/O system. We have presented an I/O model of the application depending on its I/O phases. The I/O model of application is defined by three characteristics: metadata, spatial global pattern and temporal global pattern. We instrument the application to obtain the access pattern and we analyze the pattern to find the I/O phases. This instumentation is done at MPI-IO level which does not requiere the source code. We have evaluated the I/O system utilization of different I/O configurations by considering the I/O model of application and the I/O system.

This methodology was applied in two different configurations for the NAS BT-IO benchmark and MadBench2. The characteristics of both I/O configurations were evaluated, as well as their performance usage by the application. We have extracted I/O models of applications and we evaluated the difference of performance on I/O configurations taking into account the I/O phase behavior of application in I/O devices.

As future work, we will extend the I/O phases identification to different applications which show different I/O behaviors. We are analyzing real scientific applications to obtain their I/O models. We will use the I/O model to support the evaluation, design and selection of different configurations of the I/O system.

Acknowledgment

This research has been supported by the MICINN Spain under contract TIN2007-64974, the MINECO (MICINN) Spain under contract TIN2011-24384, the European ITEA2 project H4H, No 09011 and the Avanza Competitividad I+D+I program under contract TSI-020400-2010-120.

References

[1] A. Wong, D. Rexachs, and E. Luque, "Extraction of parallel application signatures for performance prediction," in *HPCC, 2010 12th IEEE Int. Conf. on*, sept. 2010, pp. 223–230.

[2] J. H. Laros *et al.*, "Red storm io performance analysis," in *CLUSTER '07: Procs of the 2007 IEEE Int. Conf. on Cluster Computing*. USA: IEEE Computer Society, 2007, pp. 50–57.

[3] J. M. Kunkel and T. Ludwig, "Performance evaluation of the pvfs2 architecture," in *Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on*, feb. 2007, pp. 509 –516.

[4] P. Carns, K. Harms, W. Allcock, C. Bacon, R. Latham, S. Lang, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," in *27th IEEE Conference on Mass Storage Systems and Technologies (MSST 2011)*, 2011.

[5] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel i/o prefetching using mpi file caching and i/o signatures," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, nov. 2008, pp. 1–12.

[6] M. P. I. Forum, "Mpi: A message-passing interface standard," Tech. Rep., 2009. [Online]. Available: http://www.mpi-forum.org/docs/mpi-2.2

[7] J. Carter, J. Borrill, and L. Oliker, "Performance characteristics of a cosmology package on leading hpc architectures," in *High Performance Computing - HiPC 2004*, ser. Lecture Notes in Computer Science, L. Bougé and V. Prasanna, Eds., vol. 3296. Springer Berlin / Heidelberg, 2005, pp. 21–34.

[8] H. . S. J. Shan, "Using ior to analyze the i/o performance for hpc platforms," LBNL Paper LBNL-62647, Tech. Rep., 2007. [Online]. Available: www.osti.gov/bridge/servlets/purl/923356-15FxGK/

[9] W. D. Norcott, "Iozone filesystem benchmark," Tech. Rep., 2006. [Online]. Available: http://www.iozone.org/

[10] P. Wong and R. F. V. D. Wijngaart, "Nas parallel benchmarks i/o version 2.4," Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division, Tech. Rep., 2003.

Language and Debugging Support for Multi-Agent and Spatial Simulation

Niko Simonson Sean Wessels Munehiro Fukuda*

Computing & Software Systems
University of Washington, Bothell,
18115 NE Campus Way, Bothell, WA 98011

Abstract

The MASS (Multi-Agent Spacial Simulation) library facilitates parallelization of applications that are viewed as interaction among up to millions of agents behaving over a shared virtual space and that are thus fitted to simulation of ecological, social, and physical mechanisms. The library invokes user-defined functions of all agents and array elements as well as exchanges data among them in parallel. The key to success of this library implementation is to accelerate function invocation with preprocessor-generated code and to facilitate an application debugging environment. This paper presents the design strategy, implementation, and usability of the MASS library preprocessor and debugger.

1 Introduction

Multi-agent individual-based models view computation as interaction among agents and individuals, each autonomously behaving in a shared simulation environment. They have been used for years to simulate ecological, social, and physical mechanisms that are generally difficult only with mathematical formulae. To parallelize these models, we are developing the MASS (Multi-Agent Spatial Simulation) library that updates the status of all objects at once with the *callAll* method and exchanges data among all objects at once with *exchangeAll* method. These methods are attributed as (1) one-sided parallel communication from the main function to all array elements and (2) one-sided parallel communication from each element. Therefore, MASS benefits not only multi-agent models but also data-intensive computation with its parallelization.

We implemented MASS in Java from the viewpoint of its widely used and convenient graphics features. However, due to Java's nature as well as the multi-agents' behavioral complexity, MASS encounters the following four challenges: (1) parallelization is killed by the slow Java reflection that is used to identify a user function called from *callAll/exchangeAll*; (2) *exchangeAll* incurs substantial communication overhead if applied to computationally fine-grained elements; (3) a programmer needs to check inter-element communication flow at an application level; and (4) agent migration is difficult to keep track of at a user level.

To address these problems, we have developed a language preprocessor and GUI-based debugger for the MASS library. The preprocessor inserts additional code in a given application for calling a user function from *callAll/exchangeAll* without using Java reflection and for transferring multi-element data in bulk. The debugger runs between a user application and the underlying MASS library to capture all the library calls so that it graphically shows each object's status, monitors inter-object communication, keeps track of agent migration, and stops/resumes the user program at a break point.

This paper describes the preprocessor-assisted MASS performance improvement and library extension, presents the features and internal design of the MASS debugger, and demonstrates the uniqueness and usability of these software tools in comparison with related work.

2 MASS Library

2.1 Execution Model

Places and *Agents* are keys to the MASS library. *Places* is a multi-dimensional array of elements that are dynamically allocated over a cluster of multi-core computing nodes. Each element called a *place*, is pointed to by a set of network-independent array indices, and is capable of exchanging information with any other *place* objects. *Agents* are a set of execution instances that can reside on a *place*, migrate to any other *place* with array indices, and interact with other *agent* objects as well as multiple *place*s.

As shown in Figure 1, parallelization with the MASS library uses a set of multithreaded communicating processes

*Corresponding author. Email: mfukuda@u.washington.edu, Phone: 1-425-352-3459, Fax: 1-425-352-5216

that are forked over a cluster and are connected to each other through ssh-tunneled TCP links. The library spawns the same number of threads as the number of CPU cores per node. Those threads take charge of method call and information exchange among *place*s and *agent*s in parallel.

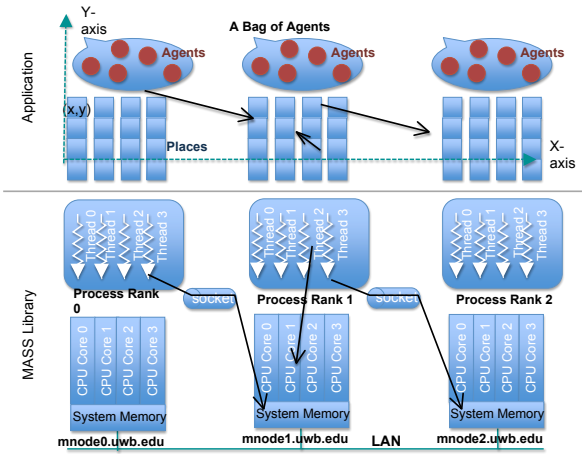


Figure 1. Parallel execution with MASS library

2.2 Library Specification

A user designs behaviors of a *place* and an *agent* by extending the *Place* and *Agent* base classes respectively, populates them through the *Places* and *Agents* classes, and performs their computation through the following methods.

Places Class

- ***public Places(int handle, [String primitive,] String className, Object argument, int... size)*** instantiates a shared array with *size* from *className* or a *primitive* data type as passing an *argument* to the *className* constructor. This array receives a user-given *handle*.
- ***public Object[] callAll(String functionName, Object[] arguments)*** calls the method specified with *functionName* of all array elements as passing *arguments[i]* to element[i], and receives a return value from it into *Object[i]*. Calls are performed in parallel among multi-processes/threads. In case of a multi-dimensional array, *i* is considered as the index when the array is flattened to a single dimension.
- ***public Object[] callSome(String functionName, Object[] argument, int... index)*** calls a given method of one or more selected array elements. If *index[i]* is not negative, it indexes a particular element, a row, or a column. If *index[i]* is negative, say $-x$, it indexes every x^th element. Calls are performed in parallel.
- ***public void exchangeAll(int handle, String functionName, Vector<int[]> destinations)*** calls from all elements a given method of all destination elements, each indexed with a different *Vector* element. Each vector element, say *destination[]* is an array of integers where *destination[i]* includes a relative index (or a distance) on the coordinate i from the current caller to the callee element. The caller passes its *outMessage[]* data member to the callee as a set of arguments, and receives return values in its *inMessage[]*.
- ***public void exchangeSome(int handle, String functionName, Vector<int[]> destinations, int... index)*** calls each of the elements indexed with *index[]*. The rest of the specification is the same as *exchangeAll()*.

Agents Class

- ***public Agents(int handle, String className, Object argument, Places places, int population)*** instantiates a set of agents from *className*, passes the *argument* to their constructor, associates them with a given *Places* matrix, and distributes them over these places, based on *map()* that is defined within the *Agent* class.
- ***public void manageAll()*** updates each agent's status, based on its latest calls of *migrate()*, *spawn()*, *kill()*, *sleep()*, *wakeup()*, and *wakeupAll()*. These methods are defined in the *Agent* base class and may be invoked from other functions through *callAll()* and *exchangeAll()*.

2.3 Design Issues

From the user viewpoint, *Places* and *Agents* are theoretically considered as an array or a collection of *Place* and *Agent* objects respectively. However, their underlying implementation is not so simple in order to not only serve as a general simulation framework but also to work over a distributed-memory cluster. We need to address the two design challenges below:

Language Issues: Unless a user implements base methods of the *Place* or *Agent* class, *Places* and *Agents* do not know any methods of a user-defined class. This in turn means that the *callAll/Some* and *exchangeAll/Some* methods cannot invoke a user function simply through object casting. Instead we need to use Java reflection to interrogate a user-defined class. The problem is that the reflection works one order slower than a direct function call in general. This slow performance kills parallelization where MASS calls the same function of all objects at once. To pursue both naming flexibility and high-speed invocation of user functions, we design a Java preprocessor that inserts additional code to match the names of user functions defined in MASS methods and the actual function bodies to invoke, so that the library calls user functions without the reflection.

Debugging Issues: Since *Places* and *Agents* may be allocated over a distributed-memory cluster, the status and execution of their elements is not always visible and traceable where the main program is running. For debugging purposes, users are responsible to collect remote element status by manually inserting combinations of *callAll/Some* and *exchangeAll/Some* methods as well as adding additional graphics code into their application programs. In particular, it is tedious work for application designers to keep track of migrating *Agent* objects over different computing nodes. We address these debugging issues by designing a wrapper that covers the original MASS library and facilitates GUI-based debugging features.

The next two sections explain these solutions.

3 Preprocessor Design

3.1 Library Extension

We extend MASS to avoid Java reflection and to accelerate message exchange among *Place* objects as follows:

Eliminating Java Reflection: Given a function name in the MASS methods, we need to quickly identify its function body to invoke. A trivial but naive idea is to store user function names in a symbol table at compile time and thereafter to compare each symbol table entry with a function name specified in a MASS method each time it is invoked at run time. The problem is repetitive string comparisons that may weigh more than actual computation at each *place* or *agent*. Instead we use integer comparisons where user function names found in MASS methods receive a different integer, (i.e., a function id) at compile time and a MASS method invokes the user function corresponding to a given function id. Figure 2 shows preprocessor-generated example code that jumps from two MASS methods to a different user function: two user function names such as "exchangeArray" and "putArray" in *exchangeAll()* and *callAll()* (lines 1-2) receive a function ID respectively (lines 6-7); the original *exchangeAll()* and *callAll()* calls the preprocessor-generated *callMethod()* (line 12); and the control branches off to the corresponding user function, based on the function id (lines 14-15).

Exchanging Each Place's Boundary Information: The MASS library originally assumes that a *Place* object is used as an individual element of a distributed array. However, the cost for *exchangeAll/Some* is substantial to fine-grained computation at each *Place* object, because data exchange generally takes place with multiple neighbors. Therefore, a user wants to include a collection of array elements in each *Place* object that then exchanges its boundary elements (or shadow elements) in fewer packets with neighbors as shown in Figure 3. This reduces the frequency of communication over an entire array while increasing the

```
1   // the original MASS methods
2   myPlaces.exchangeAll(h, exchangeArray, neighbors);
3   myPlaces.callAll(putArray, args);
4
5   // preprocessor-generated code to jump user functions
6   public static final int exchangeArrayP_ = 0;
7   public static final int putArrayP_      = 1;
8
9   myPlaces.exchangeAll(h, exchangeArrayP_, neighbors);
10  myPlaces.callAll(putArrayP_, arggs);
11
12  public Object callMethod(int funcId, Object args) {
13     switch(funcId) {
14        case exhangeArrayP_: return exchageArray(args);
15        case putArrayP_:     return putArray(args);
16     }
17     return null;
18  }
```

Figure 2. Two MASS methods and their preprocessor-generated code

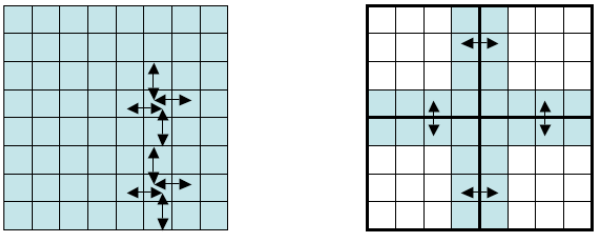


Figure 3. Communication among neighboring Place objects

computation amount per *Place*. A new MASS library function named *exchangeBulk* exchanges such boundary information among neighboring *Place*s. As shown in Figure 4, we achieve it by translating *exchangeBulk* into a combination of *exchangeAll* and *callAll* (lines 6-7): the former calls a given function of all neighboring *Place* objects to retrieve their boundary information, and the latter put the information into the local *Place*'s boundary space. The MASS preprocessor assumes that a user defines *exchangeArray* and *putArray*, where *Array* is a user-defined *Place* object, each actually achieving data retrieval and saving operations. If not, the preprocessor generates simple stub functions (lines 9-15). Thereafter, it converts this pair of *exchangeAll* and *callAll* into those calling the user functions with their function IDs as described in Figure 2.

3.2 Design Strategies

We design and implement the MASS preprocessor, based on the following two strategies. First, we use an existing Java compiler-compiler tools: JavaCC and JJTree

```
1   // A new MASS method to exchange boundary data
2   myPlaces.exchangeBulk(h, Array, neighbors);
3
4   // preprocessor-generated exchange/callAll from
5   // exchangeBulk
6   myPlaces.exchangeAll(h, "exchangeArray", neighbors);
7   myPlaces.callAll(h, "putArray", neighbors);
8
9   public Object exchangeArray(Object src) {
10     return (Object)Array.getBoundary((int[])src);
11  }
12  public Object putArray(Object arg) {
13     Array.putBoundary(inMessages);
14     return null;
15  }
```

Figure 4. exchangeBulk and its preprocessor generated code

for parsing and optimizing MASS user programs. Second, we carry out two passes of MASS program translation: pass 1 converts *exchangeBulk* into a combination of *exchangeAll/callAll*, and pass 2 generates additional code to call a user function from a MASS library method with its function ID. The following details an implementation of our MASS preprocessor.

3.3 Implementation

The preprocessor performs its optimizations by running the input code through a Java parser. The parser emits tokens in response to the input code. Actions are taken on specific tokens to check conditions, set flags, and modify output. A grammar defines a roughly correct version of Java. It has been modified to create Abstract Syntax trees. From the grammar, a parser is generated (as UnparseVisitor.java) which by default will output any input which matches the Java language as defined by the grammar. As shown in Figure 5, the parse methods can be overwritten with MASSOptimizer, ExchangeBulkOptimizer, or ReflectionOptimizer to perform MASS optimizations. For example, when parsing a *MethodDeclaration* token, a flag will be set to indicate that a new method is being parsed and that a new scope must be placed on the stack. Subsequently, if a *ResultType* token is parsed while the *MethodDeclaration* flag remains set, the return type of the parsed method can be recorded. These optimizers in Figure 5 set flags in their *visit()* methods and implement the logic to respond to those conditions in their *find(Token)* methods.

The preprocessor has been tested on some MASS programs including two-dimensional wave simulation (Wave2D) and three-dimensional computational fluid dynamics (CFD). The verification and performance evaluation has been conducted by comparing manually-translated versus preprocessor-generated code. Figure 6 demonstrated the competitive performance of preprocessor-generated code in

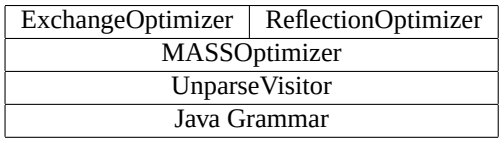


Figure 5. MASS preprocessor implementation

Code	Total	exchangeAll	callAll
Manual	9730.5 ms	4605.25 ms	1505.5 ms
Preprocessor	9785 ms	4366.25 ms	1705.75 ms

Wave2D

Code	Total
Manual	9730.5 ms
Preprocessor	9785 ms

CFD

Figure 6. Preprocessor-generated code execution

Wave2D and CFD when running the code four times on a 64-bit 2.27GHz Intel Core.

4 Debugger Design

A simple debugging program was implemented to assist MASS developers. Currently, the debugger uses the multi-threaded Java version of MASS. It allows users to view a logical arrangement of their computational spaces' values through a 2D or 3D graphical view.

4.1 Debugging Features

The basic objective of the debugger is to display the contents of computational nodes in a human-understandable format, as shown in Figure 7. Its features are designed to support this goal:

- Displays results in a flat view or hawk's eye view.
- Opens windows to display additional dimensionality.
- Allows debugging in code or in GUI.
- Sets break points and defines iteration points.
- Advances to next break point or by iteration.
- Shows communication between logical nodes.
- Saves and restores computational values.

The developer can set break points in program iteration. This differs from traditional debugging code break points, and is more akin to setting break points based on variable values. However, the developer can specify when in the driving code that an iteration occurs, allowing a more fine-grained approach than might be initially apparent.

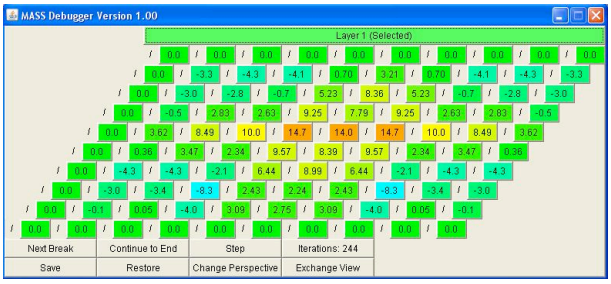


Figure 7. Debugger GUI's hawk's eye view of computational node values.

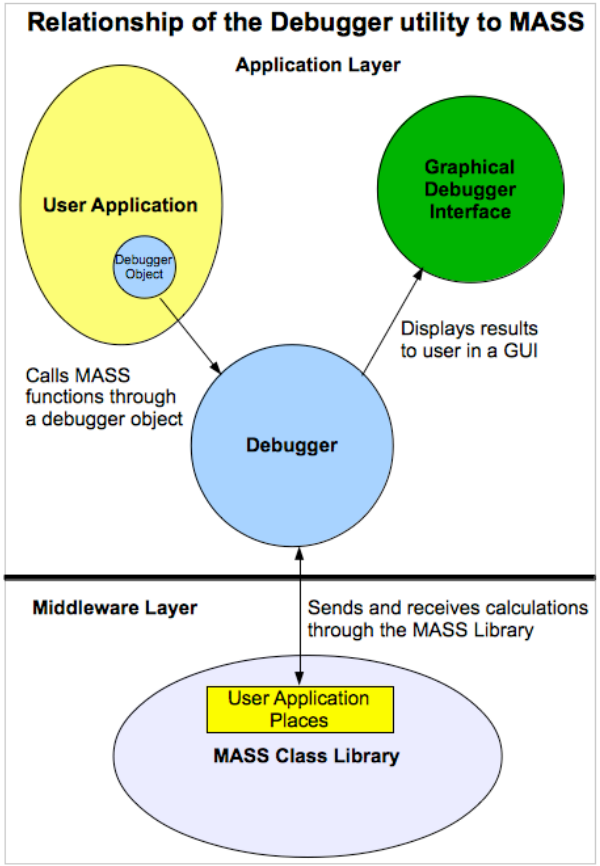


Figure 8. Interaction between the debugger, a user application, and MASS.

4.2 Design Strategies

Design of the debugger followed a twofold strategy. The debugger wraps functions of the MASS class library as shown in Figure 8, and development is driven by the need for features to show information in terms of the MASS application developer's perspective. The development of the debugger relied on an iterative approach that focused on design, planning, and prioritization of features. A basic goal is to implement features as simply as possible. Early in the process it became apparent that there were two challenges: accessing MASS and creating the graphical interface. Comparing the two, using MASS was simpler than creating the graphics, but had yet to be thoroughly explored. The debugger is the first application to demonstrate that multiple classes can utilize MASS concurrently.

4.3 Implementation

The debugger program exists separately from the MASS library. Therefore, it is not affected by changes in the MASS implementation that don't affect MASS function signatures. In effect, it serves as an intermediary layer that wraps MASS functions. In addition to the debugger class, there are classes for graphical control objects: the display windows and buttons. When data is returned from the MASS classes, in addition to returning the data to the user, the debugger also instantiates graphical objects using Java's awt classes, and displays the results. The organization of the results are based on the user-defined sizes of the arrays that are passed to MASS when it is initialized. In this context, the implementation of the debugging features only has to use simple private counters and Booleans to keep track of iterations and breakpoints. When a breakpoint is hit, further function calls to MASS are suspended. Since all MASS results are returned in single arrays, there is no latency in updating individual node results from the debugger or user perspective. Depiction of exchange results are more difficult, because MASS does not provide return values for them. The user provides a vector of relative node coordinates where data is exchanged. The debugger applies the vector to each computational node to determine the coordinates where the data exchange occurred, and shows the last computation (from a MASS *CallAll* function) as the data that is sent. Checkpoints take the return values from a computation and store them in a file. The file is read back into an array of objects when the checkpoint is restored, and passed to MASS. It is important to note that this method of restoring a checkpoint is only valid if the user's places do not rely on local resources, such as a local system clock, for their calculations. The debugger offers three general types of public methods, as shown in the sample code of a simple driver application in Figure 9:

- MASS-equivalent functions with identical signatures
- MASS-equivalent functions with extended signatures for the debugger
- Debugger-only functions, such as setting break points.

The MASS functions with extended signatures combine MASS-equivalent functions with debugger functions, trading fewer function calls for ones with more parameters. Control buttons also implement debugger functions for features such as advancing one iteration or continuing to the next break point while the user's program is running.

```
1  // Initialization of MASS through debugger
2  debugger.degubInit(totalSize, totalDimensions,
3        ''DebuggerDriver'', threads, 0, 999);
4
5  // Pure debugger functions
6  debugger.setTotalIterations(iterations);
7  debugger.setBreakPoint(20);
8  debugger.setStopOnBreakPoint(true);
9
10 // Use of MASS through debugger
11 while(!debugger.isFinished()) {
12    // MASS-equivalent function called through
13    // debugger (shows results in GUI by default)
14    debugger.debugCallAll(0, (Object[])null);
15
16    // ...with iteration set separately
17    debugger.iterate();
18
19    // MASS function with debugger parameters
20    // (ticks iterator shows results in GUI)
21    debugger.debugIterateCallAll(0, (Object[])null,
22       true);
23 }
24
25 // end debugger operations and clean up graphics
26 debugger.finish();
```

Figure 9. Code using debugger; breakpoint set for every 20 iterations

5 Related Work

5.1 Preprocessors

Our MASS library and preprocessor design involves library-assisted parallel execution, preprocessor-assisted code generation, and code manipulation. Those techniques are found in the following four language systems.

Parallel Java Library promotes hybrid SMP cluster programming in Java by combining multithreaded programming constructs and MPI-based message-passing functions [4]. MASS and Parallel Java libraries both take a similar approach in hiding all underlying parallelization work with their Java library classes and methods. However, the major difference is that MASS does not distinguish shared and distributed memory but gives a consistent view of multi-agents running on a shared array regardless of underlying memory architectures.

Extensible PreProcessor (EPP) defines plug-ins for generating tiny data-parallel Java code [3], where the special modifier *"parallel"* given to a Java class generates additional multithreading code in its *run()* method that handles all data with virtual processors in parallel. Although MASS and EPP use a preprocessor approach for parallelization, EPP focuses on multithreading, whereas the MASS preprocessor extends its scope to hybrid SMP cluster computing.

MPIPP is another preprocessor tool that converts a user-defined data structure into MPI-derived data types [6]. It is similar that our *exchangeBulk()* function converts a certain range of boundary array elements to *Place.outMessage* as well as *Place.inMessages* back to the original elements. However, our MASS preprocessor is different in generating *get()* and *put()* methods to automate entire boundary-to-boundary element transfers.

Javassist facilitates a compiler-assisted Java bytecode manipulation that defines new classes, freezes existing classes, and customizes class members [1]. Therefore, Javassist can works as another option to optimize the *exchangeBulk()* function and to match user function names in the MASS library and their actual function bodies, by directly manipulating a user program. However, it would be the same amount of work required if our preprocessor were redesigned to introspect and manipulate all MASS keywords with Javassist.

5.2 Debuggers

The MASS debugger is a framework-oriented, library-based, visualization-focused, and data-parallel application debugger. In these categories, we found similarities to and differences from the following four products.

Hadoop and the MASS debugger are both framework-oriented debugging utilities. Hadoop uses a Java class based on the JUnit3 test case and performs testing using a virtual map cluster [8]. The MASS debugger debugs through the actual execution of the MASS program, as MASS controls where its computational spaces run. Hadoop Test Case output is console output, shown through the user's IDE. MASS debugger output uses its own graphical interface.

MPI Debugging Interface is used to provide the conceptual message passing state of the program [2]. The debugger implementation studies the communicator queues; the MASS equivalent of these are the parameters of its major functions. Both applications conclude that accessing the library class must be led by the debugger. Specific interfaces to display data are not implemented by the MPI debugger in contrast to the MASS debugger's GUI.

TotalView is a feature-heavy, commercial debugging interface designed for distributed software [7]. TotalView, like MASS, makes use of a GUI for data visualization but lacks concurrent display of values. The TotalView software is more sophisticated than MASS but also more complicated, and is not free. MASS, in contrast, is geared specifically to MASS users and is currently free. Using TotalView with MASS would strip away all the abstraction that the

MASS library is providing.

Global Arrays are a means to solve data-parallel jobs, and intersects with MASS's application areas [5]. It gives the user very fine-grained control over the data objects it uses, but at the cost of a great deal of programming complexity in implementation. In contrast, MASS aims explicitly to simplify and abstract away complexity. Global Arrays do not in themselves provide explicit debugging tools; while the MASS debugger is an extension of MASS that does not currently exist in Global Arrays.

In summary, we believe that our design strategy for the MASS preprocessor and debugger fits user requirements for multi-agent spatial simulation.

6 Future Work

6.1 Preprocessor

At present our preprocessor has the following limitations and issues: (1) MASS applications to be processed should not contain a method named "*callMethod*" or use a trailing underscore ("_") as part of a method name; (2) all methods to be called from *callAll/Some* or *exchangeAll/Some* must accept the same set of parameters in the same order; (3) user-owned non-MASS methods should not have names identical to MASS methods; and (4) MASS variables may not be recognized if they are cast or assigned to other classes at runtime. We believe that these limitations are acceptable.

As our future work items, we are developing C, C++, and CUDA-C versions of the MASS library. Since C and C++ allow programmers to use dynamic linking and function pointers, we do not see any necessity of developing a MASS preprocessor for them. However, CUDA-C has its own extensions to C. Although these extensions are quite unique to GPUs, we are planning to develop a preprocessor that assists C programmers in running MASS applications on GPUs. Our ultimate goal is to facilitate a Java or C++ version of the MASS library for GPU computation through a cascading code translation to CUDA-C.

6.2 Debugger

Currently, the MASS debugger focuses on the step-by-step display of computational node data and communication. The limitations include: (1) agents and their migration are not displayed; (2) the interface is still only a proof-of-concept design; (3) break points are not based on computational node values; and (4) analysis and step-through of the developer code itself is not implemented.

Implementation of agent status and migration view is the immediate next step in terms of future development. The interface must be improved for the overall user experience. Allowing break points based on computational node values will give the user more flexibility. However, the goal of the MASS debugger is to provide an easy way to view MASS computations, not to become a commercial debugger with a full feature set. Consequently, certain debugger features: the ability to step through code and to set break points in the code itself, are not prioritized for future development.

7 Conclusions

The MASS library eases parallelization of multi-agent individual-based models as well as data-intensive applications. In this paper, we analyzed the current issues in code development and execution with the library, addressed them with our preprocessor approach and debugger design. The MASS library and tools will be made available upon an email request sent to *dslab@uw.edu*.

References

[1] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering - GPCE'03*, volume LNCS 2830, pages 364–376, Erfurt, Germany, September 2003. Springer-Verlag.

[2] J. Cownie and W. Gropp. A standard interface for debugger access to message queue information in MPI. In *Proc. of Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting - PVMMPI'99*, volume LNCS 1697, pages 51–58, Barcelona, Spain, September 1999. Springer.

[3] Y. Ichisugi and Y. Roudier. The extensible Java preprocessor kit and a tiny data-parallel Java. In *Proc. of the Scientific Computing in Object-Oriented Parallel Environments ISCOPE'97*, volume LNCS 1343, pages 153–163, Marina del Rey, CA, December 1997. Springer-Verlag.

[4] A. Kaminsky. Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java. In *Proc. 21st IEEE Int'l Parallel and Distributed Processing Symposium - IPDPS*, pages 1–8, Long Beach, CA, March 2007. IEEE-CS.

[5] M. Krishnan, B. Palmer, A. Vishnu, S. Krishnamoorthy, J. Daily, and D. Chavarria. The global arrays user manual. Technical report number pnnl-13130, Pacific Northwest National Laboratory, Richland, WA, November 2010.

[6] E. Renault and C. Parrot. MPI pre-processor: generating MPI derived datatypes from C datatypes automatically. In *Proc. 2006 Int'l Conf. on Parallel Processing Workshops*, pages 248–256, Columbus, OH, September 2006. IEEE CS.

[7] Rogue Wave Software, Inc. Totalview. User Guide Version 8.9.2, Boulder, Colorado, November 2011.

[8] J. Venner. *Pro Hadoop*, chapter 7, Unit Testing and Debugging, pages 207–237. Apress, New York, NY, 2009.

Stream Processing Approach on the Fuce System for parallelizing Nested Loops with Data Dependency

Satoshi Amamiya, Makoto Amamiya
Department of Informatics, Faculty of Information Science and Electrical Engineering,
Kyushu University

Abstract— *It is widely known that nested loops with data dependency can be parallelized by transforming the original loops by techniques such as loop skewing, loop interchange (so called software pipelining and/or wavefront method). However, these loop transformation techniques need to solve the equation of simultaneous inequalities and modify loop index variables by "human". These procedures require complicated formalities to programmers. We have developed a much simpler method to parallelize nested loops with data dependency, which is an straightforward application of asynchronous handshaking stream programming on the continuation-based multithreading.*

In this paper, we discuss our new approach in detail and show the enough performance enhancements can be exploited by our approach by evaluating the parallelized version of nested loop programs, on the Fuce runtime system on a commodity machine.

Keywords*: stream processing, multithreading, loop parallelization*

1. Introduction

The importance of high performance computing (HPC) have been widely recognized in the world and a lot of the researches on HPC have been conducted in past decades. High performance scientific computations are mostly large array operations. The speedup of array operations relies on basically the parallelization of multi-level nested loops for array. A variety of techniques for the loop parallelization was developed. In particular, for nested loops with data dependency, parallelization methods such as loop skewing, loop interchange (so called software pipelining and/or wavefront method) are noted. However, these loop transformation techniques need to solve the equation of simultaneous inequalities and modify loop index variables by "human". These procedures require complicated formalities to programmers.

We need a simpler approach to the parallelization and the programming techniques. We researched and developed the parallel architecture called Fuce[1] and its programming language CML (Continuation-based Multithreading Language) based on the idea of *continuation* and *non-interruptible thread*. The CML is designed for writing low-level event-/demand-driven concurrent processes, stream processing, mutual exclusions, memory management and so on in a unified form.

In this paper, we introduce the stream processing based on the asynchronous handshake multithreading. And then we discuss the simpler parallelization technique for nested loops with data dependency as an application of our stream processing approach. Also we describe the implementation of the Fuce runtime system for commodity machines and show evaluation results of the stream processing programs executed on the Fuce runtime system.

2. Conventional Method for Parallelization

The wavefront method[2] is known as a typical method for parallelizing multi level nested loops with data dependency. We briefly explain the wavefront method using the following sample program[3]:

```
for (i = 2; i <= 6; i++)
  for (j = 2; j <=6; j++)
    a[i][j] = (a[i-1][j] + a[i][j-1]
          +a[i+1][j] + a[i][j+1]) / 4
```

To eliminate data dependencies inside the loop, first we have to do loop-skewing and then interchange the outer and the inner loops. The transformed program looks like:

```
for (j = 4; i <= 12; j++)
  for (i = max(2,j-6);
        i <= min(6,j-2); j++)
    a[i][j-1] = (a[i-1][j-i]
      + a[i][j-1-i] + a[i+1][j-i]
      + a[i][j+1-i]) / 4
```

After a modification of this program with support of a parallel programming library, we execute the program in data parallel manner on parallel machines.

However, we have to solve a simultaneous inequality on variable i and j by hand to find the range of index variable i and j. We think this is not always a simple method. In addition, the users or programmers of such a wavefront program would not obtain high performance computation if they are not so familiar with a parallel programming library.

The idea of wavefront method itself seems to be simple and beautiful. But a implementation of a data parallel program based on this idea will become complicated. If we devise a different approach from the data parallelization, We will benefit from the basic idea of wavefront method without complicated procedures.

3. Stream Processing and the CML language

We introduce a special language called CML (Continuation-based Multithreading Language) to understand a basic behavior of stream processing. The large portion of the CML is derived from the ANSI C language. Some new grammatical constructs are designed and extended for stream processing. In this section, we give an outline of new constructs only for stream processing. The detail of the CML is described in the papers[4], [5].

3.1 The Language CML

On the stream processing, temporal data storage space is usually set up between the sender and the receiver of data. Normally it is a bounded FIFO buffer. Here, for this purpose, we define a simple buffer storage called *channel* which can store basically only one data.

Channel Struct

```
struct chan {int flag, value,
             from, to;}
```

The stream is closed if `flag==0`. A variable `value` holds a single stream data. The type of value is not only integer type, but also float or double is acceptable. We just re-define a new chan type as usage. A variable `from` holds sender process id and a variable `to` holds the receiver process id.

Process Definition

```
process P(chan *ch, ...) <2> {...}
```

`*ch` is a pointer to a channel struct. `<2>` is the number of fan-in of process P. If this notation is abbreviated. the number of fan-in is 1. `{...}`is a body of process P.

Here, the number of fan-in of a process means the number of data which the process requests to send from other processes. Therefore, Once the process is scheduled and the value of fan-in of the process is initialized, the value of fan-in is simply decreasing in each data arrival. When the value of fan-in becomes zero, the process becomes ready to run [1].

Instance Creation

```
int p = new P();
```

A data area and instance of process P are reserved.

Track-laying of Stream

Set the variable `from` in channel struct to the data sender process id and set the variable `to` to the receiver process id.

Process Invocation

When invoking a process, pass the channel name as actual parameter. And use *mode indicator* <+> for a sender process, <−> for a receiver process.
For example:

```
p(ch1<+>, ch2<->);
q(ch1<->, ch2<+>);
```

Here, the process p sends a data to the process q via channel ch1 and the process q sends a data to the process p via channel ch2.

3.2 Basic Behavior of Stream

Consider a pair of processes $\langle w, r \rangle$ connected by a channel ch and that w generates a series of values for x, and sends them to r. This stream processing can be written in CML as in Figure 1. The action of w and r is controlled by cont instructions. Here we call init-, trigger-, or ack-cont for each cont instruction according to its usage. Init-cont is issued to invoke a newly created process instance. Trigger-cont is issued to invoke or prompt to invoke the other processes. Ack-cont is issued to the other process by the process which becomes ready to receive the next data.

When W is invoked by an init-cont or an ack-cont issued by R, W puts a value of x to ch, then issues a trigger-cont to R so that R can get the value. When R is triggered by W, R gets the value of x from ch, then issues an ack-cont to W so that W can put the next value. Note that the number of fan-in for $W(R)$ is two since it requires a recur-cont plus a trigger-(ack-)cont.

It never happens that both W and R run at the same time, and that ch is accessed by both. Moreover, W and R run alternately, and it never happens that W puts values to ch in succession, and that R tries to get the next value from ch before another data is put.

This behaviour is easily understood by using a Petri net-like graph depicted in Figure 2. A token represented by a black disk corresponds to an issue of cont instruction. The token in the right arrow entering w is an init-token, the tokens in the left arrows of w and r are recur-conts.

3.3 Dynamic Growth of Stream

In the examples in section 3.2 the number of processes is small and fixed. They may be unworkable in the real world. In general, the number of processes will increase dynamically. The length of stream will grow accordingly.

Consider a process named *Create* which manages a process creation. Create process is written in CML for example as following:

```
process create(chan *ch) <2> {
    darea chan ch1;
    new RW(ch<*>, ch1<+>);
    new create(ch1<->);
}
```

Here, a storage for a variable declared as `darea` is dynamically allocated at run time. After Create process creates

[1]Processes in the CML are fundamentally equivalent to threads in the Fuce execution model.

```
process main() {
  w = new W(); r = new R();
  ch->from = w; ch->to = r;
  w(ch<+>); // w's recur-cont
  r(ch<->); // r's recur-cont
}
process W(chan *ch) <2> {
  /* computing x's value */
  ch->value = x;
  cont ch->to; //trigger-cont to r
  recur;
}
process R(chan *ch) <2> {
  x = ch->value;
  cont ch->from; // ack-cont to w
  /* using x's value */
  recur;
}
```

Figure 1: A CML code for a basic stream processing.

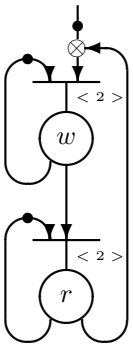


Figure 2: Petri net for a simple stream processing among two processes.

and invokes a process *RW*, it *re-creates* itself [2]. The second parameter of RW process is `ch1<+>` and the parameter of the new Create process is `ch1<->`. Therefore RW process and Create process form a stream processing among two processes as seen in Figure 1. The notation `<*>` added to the first parameter of RW process means that the channel is used for both of sending and receiving.

We define process RW in which the behavior of process W is added to process R in Figure 1. The CML code is as follows:

```
process RW(chan *in, chan *out)<3>{
  int x;
  in ?? x;   // read from channel
  out !! x;  // write to channel
  recur;
}
```

Here, '`in ?? x`' and '`out !! x`' are respectively the syntax sugar of
'`x = in->value; cont in->from;`',
'`out->value = x; cont out->to;`',
The operators `??`, `!!` are derived from CSP[6]. The number of fan-in of process RW is three because process RW is the sender as well as the receiver.

Next, in the definition of process *main* we replace the creation and invocation part of process R with them of process Create as follows:

```
process main() {
  int w = new W(ch<+>);
  new create(ch<->);
  cont w;
}
```

By this modification, process W and Create form a stream processing among two processes. And as mentioned above, process RW and Create also form a stream. When process Create is invoked, it creates process RW then pass a received data to process RW immediately. So data generated by process W are sent indirectly to process RW. Meanwhile process RW sends the data to process Create as soon as it receives. As the result as long as process W generates data, process RW is kept on being created and the size of the stream keeps on growing. Both ends of the stream is process W and Create. Between them there are myriad of process RW (refer to Figure 3).

4. Parallelization of Nested Loop

4.1 Parallelize Two-Level Nested Loop

We introduce local (instance) variable `i` to process W, `j` to process RW and Create which are defined in the section 3.3. When process W is invoked it increments the variable `i` and send the value of `i` (indirectly) to process RW via channel `ch`. Process Create increments variable `j` and creates process RW and Create with the value of `j`. Process RW keeps the value of `j` which is given at its creation.

Variables added to a parameter list of a process definition are treated as instance variables. For example, process Create is defined as follows:

```
process create(chan *ch, int j) <2>{
  darea chan ch1;
  j++;
  new RW(ch<*>, ch1<+>, j);
  new create(ch1<->, j);
}
```

Figure 3 shows the behavior of three kinds of the processes W, Create, RW which are completely moving in sync. In this figure the longitudinal direction shows passage of time, the lateral direction shows growth of the stream. Note that all these processes on the stream are moving in parallel.

[2]Note that Create process is not re-activated in tail-recursive manner by recur, it disappears after it creates another Create instance.

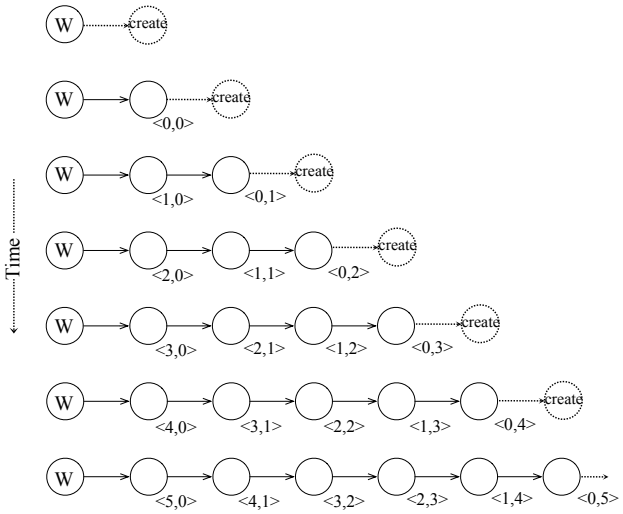


Figure 3: Ideal Stream Processing.

As mentioned above, process RW is kept on being created between process W and Create with time. The caption such like `<1,2>` in the figure is a pair of a data which is received from a immediate left process and the value of the instance variable `j` of the process RW. A data process RW receiving is initially sent by the process W, that is, the value of variable `i`.

We label the process immediate right of process W as RW_0. The initial value of `j` of process RW_0 is set to 0. A data received from process W (that is, the value of `i`) is incremented with time. The value of `j` of process RW_1 (RW_0's neighbor) is set to 1. A data RW_1 received from RW_0 is the data which the process W sent two clocks before. Therefore the value of the data RW_1 received is just as small by 1 as that of RW_o. In like wise manner, at the clock t, the pair of values process RW_n holds is `<t-n, n>`.

Now we consider the pair of values as a element of two dimensional array and observe every pair of values process RW holds at every clock. Array elements referred at same clock on the stream is completely same as the elements referred by the wavefront method[2].

That is, by using a stream processing driven by only three kinds of processes W, RW and Create, it is possible to parallelize a two-level nested loop for two dimensional array. The following loop program in C is mostly equivalent to the above stream processing:

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++) {
    ... A[i][j]...; // array access
  }
```

4.2 Parallelize Nested Loop with Data Dependency

Consider the following nested loop:

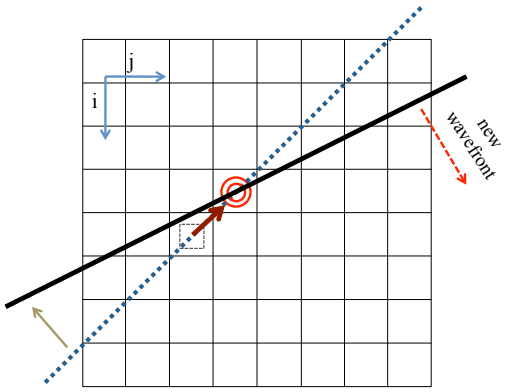


Figure 4: Wavefront on two dimensional array.

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    A[i][j] += A[i+1][j-1];
```

In this case there is a data dependency among a loop. To eliminate these kind of data dependency we have to change the direction of the wavefront movement. Figure 4 shows a movement of wavefront on a two dimensional array. The dotted line is the original wavefront. A data dependency(red arrow) exists just on the original wavefront. If the direction of the wavefront can be changed as the solid line, a data dependency disappears. To obtain the safe direction of the wavefront, we slow the speed of i-direction down by half of the speed of j-direction. When we reduce the supply of data (the value of `i`) passed on the stream by half per clock, the speed of i-direction is reduced by a half.

So, we introduce a dummy process D immediate left of process W and make process W trigger not only process RW_o but also process D. Process D triggers process W immediately when D is invoked. The number of fan-in of D is 2 and that of W is changed to 3. These modification make the speed of data supply reduced by a half because process W can run every two clocks. Figure 5 depicts the safe stream processing. Running or stopped processes is shown as white circle or gray circle respectively. A straight or curved arrow indicates a trigger-cont or a ack-cont respectively. A recur-cont is abbreviated in this figure.

While a half of processes is stopped at the same time as shown in the figure, the maximum parallelism of the stream is limited to a half number of all processes.

By the way, as we mentioned in section 3.2, in fact neighboring two processes cannot run at the same time. Therefore when we write a stream processing code naturally in CML, we get the safe stream processing like Figure 5. This is the default behavior of our stream processing approach. Such kind of data dependency disappears automatically. We don't need to care about.

Next, consider a data dependency in a loop as following:

```
A[i][j] += A[i+1][j-2];
```

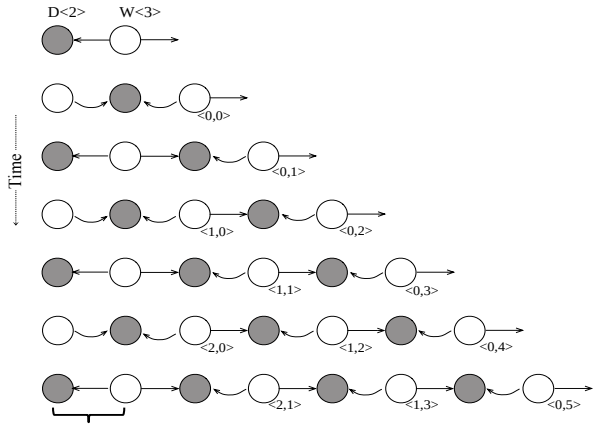


Figure 5: Stream Processing with No Data Dependency.

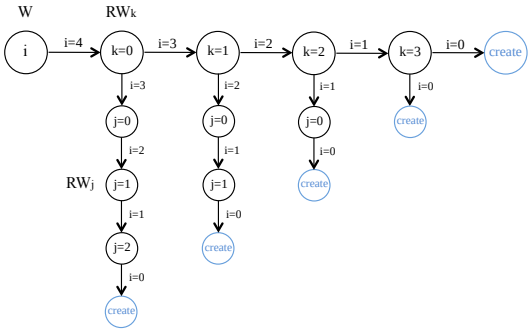


Figure 6: Stream Processing for a Three-level nested Loop.

The direction of the wavefront have to be changed also in this case. We introduce two dummy processes on left-side of process W. And each process issues ack-cont to not the neighbor process but after the next process. As a result, process W can only run every three clocks, the speed of data supply is reduced by one third. The direction of the wavefront changed to the way which satisfies $i : j = 1 : 3$.

4.3 Parallelize Three-level Nested Loop

With a small expansion of stream processing approach to a two-level nested loop, it is not so difficult to parallelize a three-level nested loop.

We modify the definition of process RW for a two-level nested loop so that process RW can act as same as process W and can make links to two different Create processes. Then two different streams will grow from a single process RW.

Figure 6 depicts a stream processing for a three-level nested Loop. A process on the crosswise stream named RW_k and that on the lengthwise stream named RW_j. The process RW_k sends the value of `i` which is passed from the process W to lengthwise and crosswise. The behavior of process RW_j is mostly as same as the process RW defined for a two-level nested loop. The wavefronts are generated as many as the number of the process RW_j and move continuously on a two dimensional array. The following three-level loop is mostly equivalent to the stream processing explained in this section:

```
for (i = 0; i < L; i++)
  for (k = 0; k < M; k++)
    for (j = 0; j < N; j++)
      ...A[i][k] = A[k][j]...;
```

By the way, if a new stream grows from RW_j, this stream processing corresponds to a four-level nested loop. In like wise, if every process on every stream has the ability to generate a new stream, a n-level nested loop can be parallelized by our stream processing approach.

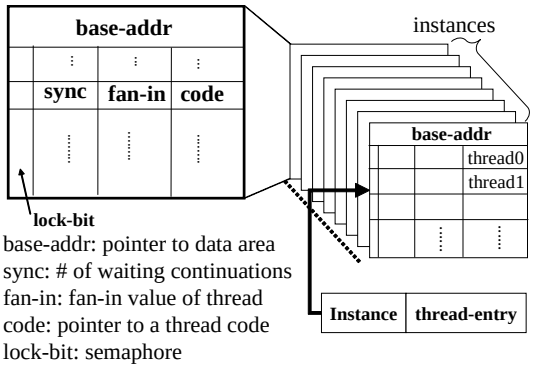


Figure 7: ACM.

5. Fuce Runtime System

We implemented the Fuce[1] processor as a software runtime on commodity OSes and parallel machines to judge the effectiveness of the idea of *non-interruptible threads* on Fuce.

5.1 ACM

The *Fuce* program is written as a set of functions. Each function is programmed using threads, and the corresponding function instances are executed in its runtime. Information of function instances is stored in a special storage called *Activation Control Memory* (ACM). Figure 7 depicts the structure of ACM. The information for controlling thread execution; sync-count, fan-in, code-entry and lock-bit, are stored in ACM.

5.2 TAC and TE

In this runtime, multiple thread execution engines are implemented. The key components of *Fuce* to support concurrent thread execution are the Thread Activation Controller (TAC), multiple Thread Execution units (TEs).

Each TE holds a ready thread queue independently. TE has the ability to *steal* a thread from the queue of another TE when its queue is empty. In order to realize this ability

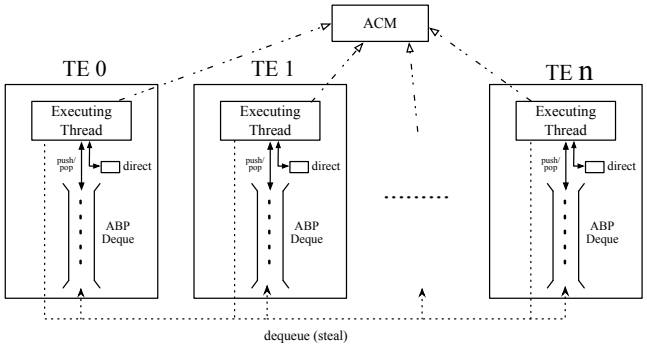


Figure 8: Multiple TEs and the ACM.

efficiently, we use ABP Deque[7] which is one of simple lock-free algorithm and can be used as a queue or a stack. With ABP Deque, a ready thread is pushed to the deque as a stack without any mutual exclusion. A thread is popped by the TE from its deque. If the deque is empty, the TE try to steal (dequeue) a thread from another TE's deque. Figure 8 shows multiple TEs accessing simultaneously to the ACM.

The main part of TE is implemented in C as following:

```
int thid;
Te_Env *env =
  (Te_Env *) GET_TE_ENV(thid);
while (!fuce_halt) {
  if (env->direct != NO_THREAD) {
    thid = env->direct;
    env->direct = NO_THREAD;
  } else {
    thid = pop(env->deque);
    if (thid == NO_THREAD)
      thid = steal_thread();
  }
  /*get thread info from ACM*/
  acm_page *page =
    &ACM[PAGE_NUM(thid)];
  thread_t *thinfo =
    &(page->entries[OFFSET(thid)]);
  void  *da  = page->darea;
  uint   id  = thinfo->id;
  Thread thr = thinfo->thread;
  thinfo->syncc = thinfo->fanin;
  /*invoke a thread as a func*/
  thr(id, da);
 }
```

Each TE can be executed concurrently on commodity OSes when the above routine is setup as a pthread.

5.3 Handling Cont Signals

a cont instruction as in Figure 1 is handled by the runtime function `cont()` with a thread id as a parameter. When the size and address of a stack for a TE (as a pthread) are properly initialized according to the pthread specification, we can determine which TE calls the function `cont()`. For example, if stack size is set to 1MB and some local variable is defined, we can obtain the stack address by the following operation:

```
#define GET_TE_ENV(ver)
    (((unsigned long) &ver)
     & (unsigned long) ~(0x100000 - 1L))
```

By using this macro the runtime function `cont()` can be simply defined as follows:

```
void cont(thid id) {
  Te_Env *env;
  int syncc;
  thread_t *target = get_thread(id);
  ATOMIC_DECREMENT(target->syncc);
  if (target->syncc < 0)
    ERROR(...);
  if (target->syncc == 0) {
    /*get current TE's env */
    env = (Te_Env *) GET_TE_ENV(id);
    if (env->direct == NO_THREAD)
      env->direct = id;
    else
      /* push thrd to ABP Deque*/
      push(env->deque, id);
  }
}
```

ACM is modified inside the function `cont()`. This modification must be done by a atomic operation because multiple threads executed in TEs would access the same location in the ACM at the same time.

6. Evaluation of Stream Processing

We evaluated the performances of stream processing CML codes on the Fuce runtime system. The program codes are transformed from two-level nested loop with data dependency. We used two benchmark programs, ISC'07 loop and the levenstein distance.

6.1 ISC'07 Loop

This sample program was created when we discovered and published[5] the basic idea of stream processing on Fuce for the first time. the program code is as follows:

```
for (i=1;i<N;i++)
  A[i][0] += A[i-1][0];
for (j=1;j<M;j++)
  for (i=0;i<N;i++) {
    if (i==0) A[i][j] += A[i+1][j-1];
    else if (i==N-1) A[i][j] += A[i-1][j];
    else  A[i][j] += A[i+1][j-1] + A[i-1][j];
   }
```

Pay attention to the second nested loop. a data dependency at the last else clause seems mostly as same type as that of example in section 4.2. We don't need to care about safety. The data dependency disappears naturally when it is transformed to a stream processing. The first loop can be fused into the inner loop of the second nested loop.

6.2 Levenstein Distance

The second benchmark program is the Levenstein distance algorithm, one of dynamic programming (DP) example.

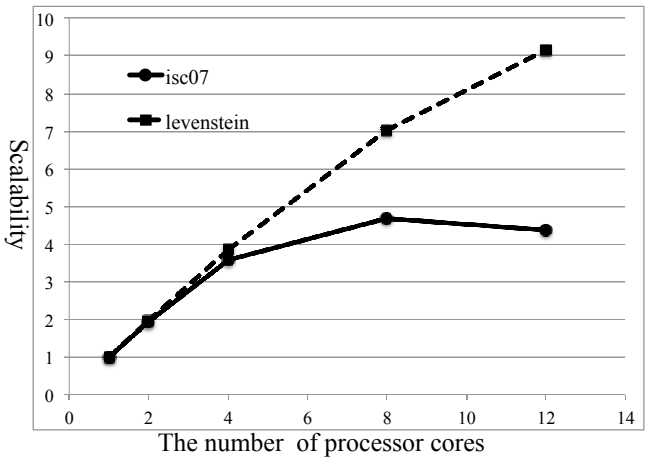


Figure 9: Scalability against the number of CPU cores.

Many other DP algorithms have similar structure to this program. the Levenstein code looks as follows:

```
int A[M+1][N+1];
for (i=0; i<M; i++) A[i][0] = i;
for (j=0; j<N; j++) A[0][j] = j;
for (i=1; i<=M; i++) {
  for (j=1; j<=N; j++) {
    int cost =
      (str[i-1] == str[j-1]) ? 0 : 1;
    A[i][j] = minimum(A[i-1][j] + 1,
            A[i][j-1] + 1,
            A[i-1][j-1] + cost); }}
```

The first two loops can be fused into the inner loop of the last nested loop. The data dependency of this program also disappears as well as that of ISC'07.

6.3 Performance Evaluation

The two transformed benchmark programs are executed on a Linux machine in which two Intel Xeon 3.3GHz processors is equipped. Each processor has six cores, in total twelve processor cores are utilized. Both transformed benchmark programs are configured so that input array size=16Kx16K, tile size=256 (256 array elements). Note that each process is defined with *tiling*[8] to operate 256 elements of array in a single process.

Figure 9 shows the scalability against the number of CPU cores for each program. On the basis of sequential execution times of the original programs, the speedups of the stream processing is shown on the graph. The performance of the levenstein is good enough. However that of ISC'07 reaches a ceiling around 8 CPU cores. We think that it is not so easy for the stream processing, especially for ISC'07 to utilize CPU caches sufficiently. Therefore the bandwidth of memory bus would be consumed away and it would interfere with communications between processes.

7. Conclusion

In this paper, we introduced the stream processing which is based on the asynchronous handshake multithreading technique and explained in detail a new approach to the wavefront method.

With our approach, it's possible to parallelize easily multi-level nested loops which hold strong data dependencies. Nested loops with data dependency appear frequently in dynamic programming algorithms. Many important time-consuming algorithms especially in biology fields are built up based on dynamic programming. We expect that the applications of our stream processing approach help to reduce time consumption of bioinformatic analyses greatly.

Acknowledgment

This research was supported by the Ministry of Education, Culture, Sports, Science and Technology(MEXT), Grant-in-Aid for Scientific Research (B), 21300011

References

[1] S. Amamiya, M. Amamiya, R. Hasegawa, and H. Fujita, "A continuation-based noninterruptible multithreading processor architecture," *J. Supercomput.*, vol. 47, no. 2, pp. 228–252, 2009.

[2] M. Wolfe, "Loop skewing: the wavefront method revisited," *Int. J. Parallel Program.*, vol. 15, no. 4, pp. 279–293, 1986.

[3] I. Nakata, *Compiler Design and Optimization (in Japanese)*. Asakura Publisher, 1999.

[4] R. Hasegawa, H. Fujita, S. Amamiya, M. Koshimura, and M. Amamiya, "Stream processing on fuce and its implementation language," *Research Reports on Information Science and Electrical Engineering of Kyushu University*, vol. 11, no. 1, pp. 31–38, 2006.

[5] S. Amamiya, R. Hasegawa, H. Fujita, M. Koshimura, R. Hirana, and M. Amamiya, "A language design for non-interruptable multithreading environment fuce," 2007, international Supercomputing Conference (ISC'07).

[6] C. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, New Jersey: Prentice Hall, 1985.

[7] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '98. New York, NY, USA: ACM, 1998, pp. 119–129. [Online]. Available: http://doi.acm.org/10.1145/277651.277678

[8] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, 1994.

Error Classifications for Parallel Message Passing Programs: A Case Study

Jan B. Pedersen[1], **Michael Jones**[1]
[1]Department of Computer Science, University of Nevada Las Vegas, Las Vegas, NV, United States.

Abstract— *Parallel programming poses additional challenges over sequential programming. In particular, parallel programming raises complexities in design, implementation, and debugging. In this work, we analyze the mistakes that student programmers make on several parallel programming problems so that we can better understand their difficulties. We administer surveys for several parallel programming projects that use MPI in two graduate level courses. The survey records the problems that the programmer had with respect to error type, and location in the parallel programming domain, the time spent on each, and the tool usage. We classify the types of errors that they encounter and find that a large percentage of the errors are sequential; these errors turned out to be the fastest to correct. We also found that the most time consuming errors to correct are associated with the messages in the message passing system. The results of the study raise implications that we can address in the classroom and through improved programming and debugging tools.*

Keywords: Parallel Programming Errors, Debugging, Multilevel Debugging, Parallel Programming

1. Introduction

Debugging parallel and multi-threaded programs can be a challenging task when programmers need to reason about multiple processes running on different physical machines and exchanging messages, sometimes asynchronously. In this work, we collect and analyze data that provides insight into how students debug these problems so that future work can build upon this understanding. Specifically, we seek to better understand the *why, how, and what of errors* in several parallel programming assignments submitted by senior and graduate students in an upper division elective course.

Research by Eisenstadt [5] and Pancake [13] has shown that upwards of 90% of programmers debugging sequential programs primarily use print statements. Whether this number has remained this high with the introduction of development environments like Eclipse [1] and xCode [2] is not currently known, but it is probably reasonable to assume that programmers who do not utilize development environments still debug using print statements. Naturally, using debuggers like gdb [8] is wide spread, but simply attaching a number of sequential debuggers, one for each process in the parallel system, is not the way to go; it leads to information overload [14], and complicates the process even further.

A natural next step is to develop tools that support efficient debugging of parallel programs. In this paper we concentrate on parallel message programs which utilize the de-facto standard for message passing, namely MPI [4]. These programs are typically written in C.

A number of debugging tools have been proposed over the years, but none have been widely adopted (Most Unix/Linux installations come standard with gdb, but no parallel version of this tool exist, and no other tool is installed by default). A number of reasons have been given for the lack of adoption of debugging tools for parallel programming [14], these include information overload, incompatible granularity, and restrictive interfaces. We believe, that the lack of adoption of these tools resonates with Pancake's reason given in [12], namely that many tools are used only by their own developers, and without taking into account the types of errors that programmer make most frequently, we will not know where to focus the tool development. That is, if we understand what kind of errors the programmers make, then we can better pinpoint the type of tool support that will make a difference. A necessary first step, before tool development, is thus to take a closer look at the types of error found, where in the programming domain they occur, and how they are found and corrected.

In this paper we explore and categorize a number of programming errors made by student programmers in a graduate class at the University of Nevada, Las Vegas. Throughout the course the programmers reported all the errors they made (through a web classification interface), they classified the errors according to location within the parallel programming domain, as well as in a 3 dimensional *why/how/what* matrix inspired by Eisenstadt [5].

We present the findings of these error reports, as well as draw parallels to a previous study we did with a different class of programmers (using the same programs). The results of the previous study [16] (henceforth referred to as *the old study* or *the old survey*) inspired us to dig deeper and ask more questions in order to get a better understanding of the parallel debugging process.

Surprisingly, the results show that a majority of the errors are in the sequential code, followed by errors in the message passing and program decomposition.

In the remainder of this paper, section 2 discusses related

work on parallel programming and parallel fault classification, section 3 describes the framework we utilized for this research, section 4 describes the programs and the survey that the programmers completed, section 6 discusses our results and section 7 provides a summary of the results and directions for future work.

2. Related Work

This section is divided into two main parts; the first part looks at the parallel programming domain, and the second part explores the error classification domain.

2.1 The Parallel Programming Domain

In this section we briefly present the *parallel programming domain* with respect to parallel message passing programs. It should be noted that we do not take aspects concerned with performance into account. While performance is extremely important, correctness is even more crucial, and that is the focus of this investigation.

We start by looking at a four stage model for constructing parallel programs (referred to as PCAM) proposed in [6]. This model consists of four parts: *Partitioning*, which is the task of partitioning both the data and functionality of the algorithm being implemented. *Communication*, which is the task of determining and implementing interprocess communication, as well as the over all protocol for the algorithm. Two other parts, *Agglomeration*, which is an evaluation of performance and *Mapping*, which is the task of maximizing process utilization, exist, but since they are performance related, they are of no importance here.

Partitioning and Communication can be further decomposed. Partitioning can be either a decomposition of the data or the functionality of the program. *Data decomposition* is the task of determining how the data (typically originating from a sequential program) should be distributed across the multiple processes of the message passing program. This included resizing arrays, implementing additional supplemental data structures etc. *Functional decomposition* is the task of determining which part of the overall computation each process should perform. An example is the implementation of a pipe-line like computation; each process is in charge of one stage of the pipe-line.

Apart from the obvious need for each process to contain sequential code, which is injected with communication calls, the communication task can be decomposed into two further sub-categories. The first is *Data Exchange* (later on referred to as *Messages*), which in a message passing program means explicitly programming send and receive calls into the code. Typically these calls are point-to-point communication involving one sender and one receiver. Since most message passing programs utilize buffered asynchronous message passing, messages can be sent to an incorrect receiver, or to non-existent receivers, without causing any run-time errors; naturally this will cause the program to stop working at some point when messages are not properly received. Stray messages can be caused by incorrect use of the message passing API, or it could be caused by a problem with the overall communication scheme, that is, the protocol. The second sub-category is *Protocol Specification*, where we consider not just point-to-point communication, but the overall communication scheme.

The two sub-categories data exchange and protocol specification, along with the obvious sequential nature of the actual code running in each process, clearly make up an onion-like structure consisting of three layers: At the core is the *sequential level*, which is responsible for the actual calculations. The second layer is the *message level* (earlier referred to as Data Exchange), which is concerned with the point-to-point communication between two processes, and finally the outer layer is the *protocol level*, which we just described.

We now have a model consisting of two main categories: *Decomposition* (data and functional) and *implementation* (sequential, messages, and protocol), which means that we have five 'buckets' into which we can categorize errors.

This categorization can help us pin-point the overall place in the development process where the most errors are made, which in turn can help us better develop tools that target the specific category. It should be noted that developing tools for decomposition errors might be a challenging task. If the design of the program is incorrect, no amount of debugging will make it right; if the data is incorrectly distributed no amount of debugging will save such a program. Decomposition errors include design flaws that must be rectified at the time the algorithm is designed.

2.2 The Why, How and What of Errors

Eisenstadt describes in [5] a 3-dimensional space in which sequential errors are placed according to certain criteria. The 3 dimensions are:

- **Dimension 1**: Why is the error difficult to find?
- **Dimension 2**: How is the error found?
- **Dimension 3**: What is the root cause of the error?

Eisenstadt's original survey surveyed 51 programmers about errors they encountered in sequential programming, but we feel the 3 dimensions are equally applicable to the parallel programming domain. The numbers reported in the following paragraphs are the numbers reported by Eisenstadt.

Unfortunately we do not have enough space in this paper to thoroughly go through the subcategories for the 3 dimensions (there are 6, 5, and 10 respectively), but it is worth noting that almost 30% of dimension 1 (why was the error hard to find) is categorized as the *Cause/effect chasm* problem. (Section 3 lists the categories, but we refer to [5] for a more thorough explanation). What describes errors in this category is the fact that the symptom of the error is far removed in space and time from the root cause. This category is bound to be an issue in the parallel

programming domain where this separation of cause and effect is further exacerbated by message passing; now the cause and effect can be in different processes, or even in different source code files. The second most frequent answer was *Tools inapplicable or hampered*, which covers the so called 'Heisen bugs' [9]. It is notable that over 50% of the cases are caused by these two categories.

In dimension 2, concerned with how an error was found, the most frequent answer was *Gathering data* (53% of answers fell in this category). This category covers the use of print statements, debuggers, break points etc. The second most frequent answer was *Inspeculation*[1], which covers hand simulation and thinking about the code. 25.5% of answers fell in this category.

An interesting, but not surprising, result is that data gathering (e.g., print statements) and hand simulation account for almost 78% of the techniques reported in locating errors (in Eisenstadt's study). This result corroborates the result of Pancake [13]: up to 90% of all sequential debugging is done using print statements.

While the use of print statements is straightforward when working with sequential programs, their use in parallel programs is more complicated. Often, processes run on remote processors, which makes redirecting output to the console difficult. Even when output can be redirected to the console, all processes are writing to the same window, thus making the interpretation of the output a challenging task. This is an example of the information overload theory mentioned earlier. Furthermore, the order of the output (i.e., the debugging information from the concurrently executing processes) is not the same for every run, as the processes execute asynchronously and only synchronize through message passing. A possible solution is to have each process write its output to a disk file. However, this introduces the problem of non-flushed file buffers; if a process crashes, the buffer might not be flushed, thus missing output written by the program. Of course this can be solved by inserting calls to flush the I/O buffers, but if these are missing, the programmer ends up spending time on debugging the code he added for debugging purposes! In the worst case this can lead the programmer to believe that the process crashed somewhere between the last print statement that appears in the file, and the first one that does not. A lot of time can then be wasted looking for an error in a place where no error can be found.

The third dimension, the root cause of the error, contains 10 different categories; the most noteworthy is the most frequent one, *Memory*, which covers errors such as overwriting a reserved portion of the memory causing the system to crash, and array subscripts out of bounds, pointer errors etc. 25.5% of answers fell on this category. The second most frequent root cause was faulty hardware (the Vendor category) (with 17.7%) and in third and fourth place, with 13.7% and 11.8% respectively, came faulty design logic (Algorithmic design/implementation problems) and initialization, which covers wrong types, redefinition of the meaning of system keywords, or incorrectly initialization of a variable.

Nearly 50% of the errors are caused by the first two categories. This also perfectly agrees with previous studies where tools and runtime systems are described as a source of errors [13]. The classification used in dimension 3 is a mixture of deep plan analysis [10], [17] and phenomenological analysis [11]. Deep plan analysis states that many bugs can be accounted for by analyzing the high level abstract plans underlying specific programs, and by specifying both the possible fates that a plan component may undergo (i.e., missing or misplaced). An alternative phenomenological taxonomy can be found in [11] where the root causes are divided into nine categories.

If we accept the decomposition of the parallel programming domain as we stated it above, as well as the overall debugging technique of hypothesis development and verification (as described in [3]), we still need to gather information about the error types like Eisenstadt did for sequential errors. This is the study presented in the following sections.

3. The Framework

In out first attempt at classification of parallel message passing errors [16], we established the usability of the PCAM modelc̆iteFo95, the decomposition of the parallel programming domain as well as introduce the 3-D model of Eisenstadt. We also reported a number of results, some of which we will recap at the appropriate places in the result section of this paper. We stated in that paper that:

> "The main goal of this research is to clarify a number of subjects related to parallel programming and debugging of parallel programs. First of all, we wish to obtain some insight into the types of errors the programmers encounter, and secondly obtain data about the techniques they used to locate and correct them. We believe that this information serves as a good basis for how programming and debugging tools for parallel (message passing) programs should be developed. It is important to understand the programming domain (in this situation, the parallel programming domain with message passing) in order to make qualified decisions about how to correct the errors."

This naturally still stands as the overall goal, but in addition, we would like to "fine tune" the survey using the 3 dimensions of Eisenstadt. We used the same assignments in this second survey as we did in the old survey described in [16], which we believe will make the results comparable.

Naturally, the end goal should be proper tool development to support debugging and error finding/correction in

[1] See section 4 for an explanation of this made-up word.

parallel message passing programs; we believe that the kind of research described in this paper can help alleviate the biggest problem with current debugging tools, as described by Cherri Pancake in [12]: tools for parallel programming and debugging are often only used by their developers. She claims that this is caused by the fact that the tool developer and the tool user might have different foci on what they want/need from a tool.

4. The Error Reports

The programmers completed a web-based questionnaire to record the run-time errors they encountered throughout the semester. The web-based questionnaire contains seven questions:

1) Why was the error hard to find? (pick one of the following): 1.) Cause/effect chasm. 2.) Tools inapplicable/hampered. 3.) What You See is Probably Illusory, Guv. 4.) Faulty Assumption. 5.) Spaghetti code. 6.) Other.
2) How did you find the error? (pick one of the following): 1.) Gather data[2]. 2.) Inspeculation. 3.) Expert recognized cliche. 4.) Controlled experiment. 5.) Other.
3) What was the root cause of the error? (pick one of the following): 1.) Memory. 2.) Vendor. 3.) Design logic. 4.) Initialization. 5.) Variables. 6.) Lexical. 7.) Unsolved. 8.) Language. 9.) Behavior. 10.) Other
4) Did you mostly use print statements?
5) Did you use any debugging tools?
6) Place the error in one of the following categories: 1.) Sequential error (including API errors). 2.) Message error. 3.) Protocol error. 4.) Data decomposition error. 5.) Functional decomposition error. 6.) Other.
7) How long did it take to correct the error?

Questions 1, 2, and 3 correspond to Eisenstadt's 3 dimensions *why, how* and *what*. Each of the sub categories for the three first questions were equipped with a more detailed explanation. As an example, this is the explanation for Inspecuation: "This term covers inspecting the code, hand simulation, and speculation. Speculation involves leaving the code to think about the problem, then later returning to try to correct it." Questions 4 and 5 pertain to tool usage and the use of print statements as a primary debugging tool. Question 6 is concerned with the placement of the error in the decomposed parallel programming domain, and finally, question 7 deals with the time spent debugging.

4.1 The Programs

In this section we briefly describe seven programming assignments that they completed during the course of the semester. These programs are (listed in the order in which the assignments were given):

- **Mandelbrot** — Use one master and n slaves to compute a Mandelbrot set.
- **Differential Equation Solver** — Solve a differential equation using a discrete method.
- **Equation Solver** — Use one master and n slave processes to solve an upper triangular system of equations.
- **Partial Sum** — Implement a partial sum algorithm that runs in time $O(log\ n)$.
- **Pipeline Computation** — Use functional decomposition to implement a multistage pipeline with dispersers and collectors that allow for multiple instances of stages of the computation to achieve a good load balance.
- **Matrix Multiplication** — Implement the Pipe-and-Roll matrix multiplication algorithm [7].
- **A project** — This could be any parallel message passing program of the programmer's choice.

4.2 Programming Domain Decomposition

We classify the types of errors from the survey questions into six overall categories; this mirrors the approach taken in the first survey [16]. These six categories are based on the first two categories of the PCAM model (partitioning and communication) as well as the three levels of the parallel programming domain. For completeness, we have added an *other* category for errors that the programmers could not place in any of the other categories. The list of the six categories are as follows:

- **Data Decomposition** — The root of the bug had to do with the decomposition of the data set from the sequential to the parallel version of the program.
- **Functional Decomposition** — The root of the bug was the decomposition of the functionality when implementing the parallel version of the program.
- **Sequential Error** — This type of error is the type we know from sequential programs. This includes using = instead of == in tests etc.
- **Message Problem** — This type covers sending/receiving the wrong data, that is, it is concerned with the content of the messages, not the entire protocol of the system.
- **Protocol Problem** — This error type is concerned with stray/missing messages that violate the overall communication protocol of the parallel system.
- **Other** — Bugs that do not fit any of the above categories are reported as 'other'. This include wrong permissions on programs to be spawned, faulty parallel I/O etc.

5. Result of Error Reporting

In this section we consider the results obtained from the error reports. First we consider the replies to question 6 (see

[2]This category has six sub categories: step-and-study, wrap-and-profile, print-and-peruse, dump-and-diff, conditional break and inspect, specialist profile tool. Note, print-and-peruse is the classic 'debugging with print statements'.

subsection 4.2) (Where in the decomposed domain did the error fall), in the next section, we look at the answers to questions 1, 2 and 3 (corresponding to the 3 dimensions from Eisenstadt's matrix). Section 5.3 looks at the debugging time (question 7), and finally in section 5.4 we conclude with the results of questions 4 and 5 pertaining to debugging using print statements and tool usage.

5.1 Error Categorization According to Domain Decomposition

Table 1 summarizes the results of the online error reporting survey and figures 1 and 2 illustrate the results in two different ways: figure 1 shows the number of errors for a given category for each of the programs (e.g., for the partial sum algorithm, 11 errors fell in the sequential error category). Figure 2 illustrates what the percentage of errors within each program fell in which category (e.g., 68.75% of the errors reported for the partial sum algorithm fell in the sequential error category). The percent number in square brackets in table 1 are the corresponding percentages from the old survey [16].

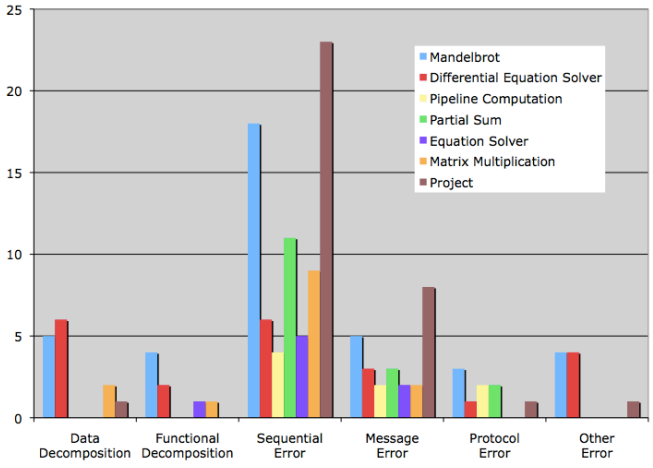


Fig. 1: Number of errors for each program categorized according to the parallel programming domain decomposition.

For the new survey, 19.63% of the errors were associated with the decomposition of the problem (for the old survey, this number was 25.16%), and 72.90% (versus 69.68% in the old survey) fell into one of the three parts of the parallel programming model. The *Other* was 7.48% (5.16% in the old survey). These results show, that the overall results from the old survey in [16] compare favorably to the result of the new survey. Now, let us look at the distribution of the errors within the parallel programming domain: In the old survey, the sequential errors made up 34.84% and in the new survey this number increased by approximately 15% to 49.53%. The message errors went from 10.32% to 15.89% and the protocol errors from 24.53% to 7.48%. (These numbers are the results excluding the project). We see that the sequential

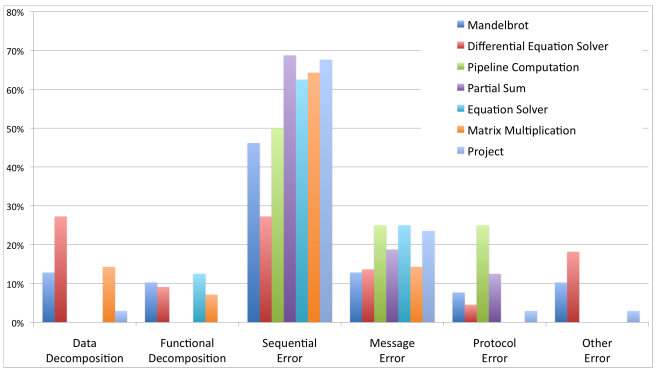


Fig. 2: Percent of errors for each program categorized according to the parallel programming domain decomposition.

errors have gone up a fair amount, and if we look at the results for the project (where no assignment or guidance was given), the sequential errors makes up a 67.63% of the errors here.

These overall results lead us to believe that support for debugging parallel programs should be focused on the sequential errors. It seems that the majority of errors (from the parallel programming domain decomposition) fall into the sequential error category; naturally support at this level must be tailored according to the already known issues such as information overload issues and granularity mismatch etc.

The old survey also reached the conclusion that sequential debugging was important, but one thing that we believe this survey revealed by including the project, is that when no guidance or development advice is given, that is, when the programmer must develop the algorithm and implement it himself from scratch, the number of sequential errors rises.

5.2 Eisenstadt's 3 Dimensional Error Matrix

The extension of the new survey was to include the categories from Eisenstadt's survey of sequential errors (Questions 1, 2, and 3). Table 2 shows the results of this categorization. The most frequently chosen category for why the error was hard to find was the *other* category (44%); unfortunately that does not tell us much. The second category is not surprisingly the *cause/effect chasm* with 23.4%; that was not unexpected as the introduction of message passing can spread the cause and the effect of an error over multiple files. These errors can greatly benefit from a tool that lets the users inspect the messages that flow from process to process. The third most reported category is the *What You See is Probably Illusory, Guv.* with 14.9%. This category covers the errors where the programmer misinterprets part of the code, that is, it does something other than he thought it would do. The fourth most reported is the *Faulty Assumption* with 13.5%. This category covers issues like assuming the stack grows down when it in fact grows up!

The most frequently chosen category for how the error was found was *gather data* with 48.2%. This category covers

Table 1: Result of the the online error reporting from (new) survey (Results from old survey in [16]).

	Data Decomposition	**Functional Decomposition**	**Sequential Error**	**Message Error**	**Protocol Error**	**Other**
Mandelbrot (n_1=39)	5 12.82[20.41]%	4 10.26[14.20]%	18 46.15[32.65]%	5 12.82[10.20]%	3 7.69[14.20]%	4 10.26[8.16]%
Differential Eq. (n_2=22)	6 27.27[8.57]%	2 9.09[0.00]%	6 27.27[48.57]%	3 13.64[11.43]%	1 4.55[25.71]%	4 18.18[5.71]%
Pipeline Comp. (n_3=7)	0 0.00[0.00]%	0 0.00[25.00]%	4 50.00[50.00]%	2 25.00[0.00]%	2 25.00[25.00]%	0 0.00[0.00]%
Partial Sum (n_4=16)	0 0.00[7.69]%	0 0.00[7.69]%	11 68.75[38.46]%	3 18.75[11.54]%	2 12.50[30.77]%	0 0.00[3.85]%
Equation Solver (n_5=8)	0 0.00[20.00]%	1 12.50[0.00]%	5 62.50[12.50]%	2 25.00[12.50]%	0 0.00[37.50]%	0 0.00[12.50]%
Matrix Mult. (n_6=21)	2 14.29[27.27]%	1 7.14[9.09]%	9 64.29[24.24]%	2 14.29[9.09]%	0 0.00[30.30]%	0 0.00[0.00]%
Project (n_7=34)	1 2.94%	0 0.00%	23 67.65%	8 23.53%	1 2.94%	1 2.94%
Total w/ Project (n = 141)	**14** **9.93%**	**8** **5.67%**	**76** **53.90%**	**25** **17.73%**	**9** **6.38%**	**9** **6.38%**
Total w/o Project (n = 107)	**13** **12.15[16.77]%**	**8** **7.48[8.39]%**	**53** **49.53[34.84]%**	**17** **15.89[10.32]%**	**8** **7.48[24.53]%**	**8** **7.48[5.16]%**

Table 2: The result of categorizing the errors according to Eisenstadt's three dimensions.

Program		**Cat. 1**	**Cat. 2**	**Cat. 3**	**Cat. 4**	**Cat. 5**	**Cat. 6**	**Cat. 7**	**Cat. 8**	**Cat. 9**	**Cat. 10**
Equation Solver	Why	2(25.0%)	0(0.0%)	1(12.5%)	0(0.0%)	0(0.0%)	5(62.5%)				
	How	3(37.5%)	3(37.5%)	1(12.5%)	0(0.0%)	1(12.5%)					
	What	1(12.5%)	0(0.0%)	1(12.5%)	0(0.0%)	1(12.5%)	0(0.0%)	0(0.0%)	0(0.0%)	5(62.5%)	0(0.0%)
Mandelbrot	Why	11	0	4	10	0	14				
	How	15	11	7	0	6					
	What	5	0	8	7	9	0	1	1	8	0
Matrix Mult.	Why	0	0	5	0	0	9				
	How	5	5	1	1	2					
	What	2	0	5	0	1	0	0	0	6	0
Partial Sum	Why	4	0	2	1	0	9				
	How	9	6	0	0	1					
	What	1	0	5	0	3	1	0	1	5	0
Pipeline Comp.	Why	3	0	1	1	1	2				
	How	4	3	0	0	1					
	What	0	0	4	0	1	0	0	1	2	0
Differential Equation	Why	7	1	0	7	2	5				
	How	8	4	1	4	5					
	What	5	0	3	3	5	0	2	0	4	0
Project	Why	6	2	8	0	0	18				
	How	24	7	1	0	2					
	What	6	0	12	1	9	0	1	0	5	0
Total	**Why**	**33(23.4%)**	**3(2.1%)**	**21(14.9%)**	**19(13.5%)**	**3(2.1%)**	**62(44.0%)**				
	How	**68(48.2%)**	**39(23.7%)**	**11(7.8%)**	**5(3.6%)**	**18(12.8%)**					
	What	**20(17.5%)**	**0(0.0%)**	**38(33.3%)**	**11(9.7%)**	**29(25.4%)**	**1(0.9%)**	**4(3.5%)**	**3(3.5%)**	**35(30.7%)**	**0(0.0%)**

the use of print statements, debugging with break points etc. Second was *inspeculation* with 23.7%. Inspeculation is an amalgamation of inspection and simulation and covers for example hand simulation of the code. The *other* category made up 12.8%.

Design Logic was the most frequently chosen category for what caused the error (it was chosen 33.3% of the time), and it basically means incorrect algorithm implementation. *Behavior* came second with 30.7%, and this category covers for example users using the program in an incorrect way. *Variables* (covering incorrect use of operators and operands) with 25.4% and *Memory* with 17.5%. This category covers pointer errors, array out of bound errors etc. In the following subsection we briefly consider debugging time.

5.3 Debugging Time

Both in the old and in the new survey we asked the programmer to report the amount of time spent on debugging.

Table 3 shows the results of this question. Note, the standard deviation is only available for the new survey. In the old survey, the average time for finding and correcting a bug was 51 minutes, in the new survey the average time was 41 minutes.

One error report reported one error taking 840 minutes (this was in the Project category); this might be a mistake. Not taking this one error into account, table 4 shows the the average time it took to correct an error; it also shows the total amount of time spent in the different categories as well as a percent-wise break down of how much time was spent in the different categories in total.

Table 4 shows that the average time for finding and correcting a sequential error was approximately 24 minutes (in the old survey, his time was 37 minutes), but it also shows that a total of 1,846 minutes was spent on finding sequential errors and that made up almost 38% of the time (counting a total of 76 out of the 141 errors) spent debugging in total.

Although the sequential errors take the least amount of time to fix per error, a majority of time is spent on this particularly category; a total of almost 38% of debugging time is spent on finding and correcting sequential errors. It should also be noted that the message errors on average take more than twice as long to correct (61.4 minutes per error), and a total of 31.14% of the overall debugging time is spent on correcting these errors.

5.4 Tool Usage and Debugging-by-print-statements

Questions 4 and 5 cover the tool usage and debugging by the use of print statements. Table 5 tabulates the result of these two questions. As expected, the number of replies in the affirmative for tool usage is almost non-existing, and the percentage for print statements as the primary debugging tool is very high; at almost 86% it matches the numbers reported by Pancake for sequential programs. This is not unexpected, but it is disconcerting that a poor debugging technique from sequential debugging has been adopted in the parallel programming domain. The main reason for this is of course the lack of tools, something we hope to help alleviate.

6. Results

The results of the surveys show that a large number (76) of the errors are sequential in nature. Even though the sequential errors are amongst the quickest to correct, the overall time spent on finding and correcting the sequential errors (which count more than 3 times as many as the second most frequent category) is in the vicinity of 40%. In addition, the message errors are reasonably frequent and they take a lot longer to correct (more than twice as long as the sequential errors). It should be mentioned that approximately 16% of debugging time (and approximately 16% of the errors) is found at the decomposition level; but as stated earlier, these are hard to develop tool support for as they are errors that have to do with the algorithm rather than the implementation.

Interestingly, the category that rose to near the top in Eisenstadt's survey for what caused the error, namely the vendor category covering hardware and software errors scored 0% in our survey; this is perhaps not strange, as hardware and compiler software is extremely stable.

7. Conclusions and Future Work

This work presents the results of a survey concerning errors made by student programmers in a parallel programming course in which they implement a number of parallel message passing programs. We contrast the results obtained in this survey with a similar survey done in 2006, but also expand the survey to include a 3 dimensional matrix suggested by Eisenstadt into which the errors were categorized. The results show that the most frequent error types are sequential errors, whereas, those that take the longest to locate and correct are the message errors. Future tool development should focus on these errors because they are frequently made and take a long time to find and correct.

Within the sequential category we see that cause/effect chasm contributes to a large percentage of the errors, and interestingly enough, these errors are exacerbated further by the introduction of message passing, which again points to necessary tool support for message errors. The most frequently applied techniques for locating the error was gathering data and inspeculation (account for almost 72% of the answers), and the error was most frequently caused by *behavior*, *errors in design logic*, *variables* and *memory* issues.

The results motivate us to pursue the development of debugging tools for parallel programs. In particular, a multilevel debugging approach [15] may help programmers to more efficiently identify and correct the errors; Multilevel debugging is tailored to focus on the decomposition of the parallel programming domain as explained in section 2.

Table 3: Average debugging time and number of responses.

Program	Average Time in Minutes		# Responses		Standard Deviation
	New	Old	New	Old	
Equation Solver	73	43	8	8	122
Mandelbrot	34	45	39	49	37
Matrix Multiplication	31	47	13	33	47
Partial Sum	45	63	16	26	52
Pipeline Computation	12	28	7	4	10
Differential Equation	42	61	22	35	90
Project	49	—	34	—	142
Overall	**41**	**52**	**139**	**155**	

Table 4: Debugging Time.

	Data Decomp.	Functional Decomp.	Sequential Error	Message Error	Protocol Error	Other
Average Time	19.9	68.1	24.3	61.4	50.0	30.6
Total Time Spent	278	545	1,846	1,536	451	245
# Errors	14	8	76	25	9	8
Total Time Spent in %	5.67%	11.12%	37.67%	31.34%	9.20%	5.0%

Table 5: Tool usage and debugging by print statements.

	Tool used for debugging	Print statements primary debugging tool
Yes	1 (0.70%)	121 (85.8%)
No	137 (97.2%)	19 (13.5%)
N/A	3 (2.10%)	1 (0.70%)

References

[1] Eclipse - The Eclipse Foundation open source community website. http://www.eclipse.org.

[2] Xcode 4 - The Apple Developer Website . https://developer.apple.com/xcode/.

[3] K. Araki, Z. Furukawa, and J. Cheng. A General Framework for Debugging. *IEEE Software*, pages 14–20, May 1991.

[4] J. Dongarra. MPI: A Message Passing Interface Standard. *The International Journal of Supercomputers and High Performance Computing*, 8:165–184, 1994.

[5] M. Eisenstadt. My hairiest bug war stories. In *The Debugging Scandal and What to Do About It - Communication of the ACM*. ACM Press, April 1997.

[6] I. Foster. *Designing and Building Parallel Programs: Concepts and tools for parallel software engineering*. Addison Wesley, 1995.

[7] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving problems on concurrent processors. General techniques and regular problems*, volume 1. Prentice Hall International, 1988.

[8] Gdb - The GNU Debugger . http://www.gnu.org/directory/gdb.html.

[9] J. Gray. Why do Computers Stop and What Can be Done About it? *Proc. of 5th Symposium on Reliability in Distributed Software and Database Systems*, January 1986.

[10] W. L. Johnston. An Effective Bug Classification Scheme Must Take the Programmer into Account. *Proc. of the workshop of High-level debugging. Palo Alto, CA*, 1983.

[11] D. E. Knuth. The Errors of TEX. *Software - Practise and Experience*, 19(7):607–685, July 1989.

[12] C. M. Pancake. Why Is There Such a Mis-Match between User Need and Parallel Tool Production? Keynote address, 1993 Workshop on Parallel Computing Systems: A Dialog between Users and Developers, April 1993.

[13] C. M. Pancake. What Users Need in Parallel Tool Support: Survey Results and Analysis. Technical Report CSTR 94-80-3, Oregon State University, June 1994.

[14] C. M. Pancake et al. Results of User Surveys Conducted on Behalf of Intel Supercomputer Systems Division, Two Divisions of IBM Corporation, and CONVEX Computer Corporation, 1989-1993.

[15] J. B. Pedersen. *MultiLevel Debugging of Parallel Message Passing Programs.* PhD thesis, University of British Columbia, 2003.

[16] J. B. Pedersen. Classification of Programming Errors in Parallel Message Passing Systems. In *Proceedings of Communicating Process Architectures 2006 (CPA'06)*. IOS Press, September 2006.

[17] J. C. Spohrer, E. Soloway, and E. Pope. A Goal/Plan Analysis of Buggy Pascal Programs. *Human-computer Interaction*, 1(2):163–207, 1985.

File Composition Technique to Improve the Performance of Accessing a Number of Small Files

Yoshiyuki Ohno[1], **Atsushi Hori**[1], **and Yutaka Ishikawa**[1,2]
[1]RIKEN Advanced Institute for Computational Science, Kobe, Hyogo, Japan
[2]The University of Tokyo, Tokyo, Japan

Abstract— *One of the scalability issues in parallel applications, in which each process creates each file and writes data to the file, is the scalability of file management due to the increasing number of files. To mitigate this issue, a new file aggregation mechanism, called the file composition technique, is proposed. Unlike existing aggregation mechanisms, the file composition technique aggregates multiple files created by parallel processes into a single shared file without changing the code of file I/O operations. In contrast with the metadata operations in existing aggregation mechanisms, the metadata operations are distributed to each process in order to carry out scalability. The proposed file composition technique is evaluated using a climate simulation code, called SCALE. The result shows that the elapsed time of file output is approximately 30% faster than that of original POSIX I/O functions.*

Keywords: File I/O, Parallel file system, File aggregation

1. Introduction

I/O access patterns of parallel applications are broadly classified into three patterns: N-1, N-N, and N-M. The N-1 pattern means that all processes access some shared files, and the N-N pattern means that each process accesses some individual files. For example, in a climate simulation program, all processes save their local data to a single file or save each individual file. In the former case, the file I/O access pattern is called N-1 and is called N-N in the later case. Contrasting with applications employing the N-1 or N-N pattern in other parallel applications, such as data mining and processing huge sensor data, all processes following the N-M I/O pattern handle many files whose sizes are not uniform.

I/O access patterns of many parallel applications are categorized as the N-N pattern. For example, it was reported that, in ten projects using the Blue Gene/P at the Argonne National Laboratory [1], four, seven, and two projects employ the N-1, N-N, and N-M patterns, respectively. In the N-N pattern, as the number of processes increases, the number of files grows and the I/O performance becomes the bottleneck of scalability. For example, Figure 1 shows the execution time of parallel processes creating one individual file each. The execution time linearly increases according to the increasing number of processes. This is the result of using a cluster with the Lustre file system described in Section 4.

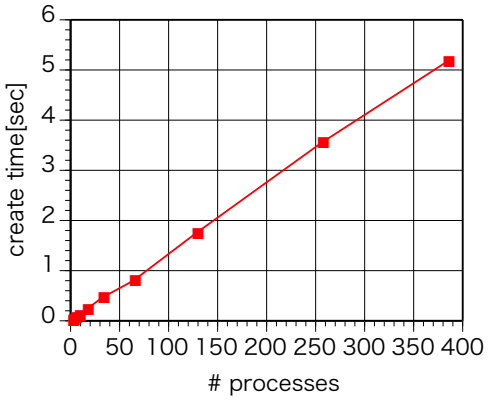


Figure 1: Time it takes parallel processes to create one file each

An approach to mitigating the above problem is introducing a file aggregation mechanism that gathers multiple data generated by an application and stores these data into one or a few files in order to reduce the number of files accessed by these processes. Several libraries carry out such a file aggregation mechanism, e.g., MPI-IO [2], PnetCDF [3], and SIONlib [4]. All of them assume that parallel applications are based on the single program multiple data 'SPMD' execution model, that is, all processes access files at the same time. Those libraries have two main issues, If an application is written using Posix file I/O APIs, it must be rewritten using their APIs. In the case that an application is not written in the SPMD manner, its modification cost is much higher. The other issue is the metadata management of libraries such as PnetCDF and SIONlib. Because the metadata of files is sequentially handled at the user level, metadata operations, such as creation and extension of files, limits scalability.

In this paper, a new file aggregation mechanism called the file composition technique is proposed. It makes application programmers select the I/O pattern such that each process may access multiple individual files. In the proposed technique, the middleware gathers files created in the application and stores them into a single shared file in a parallel file system. The Lustre file system [5] is currently utilized.

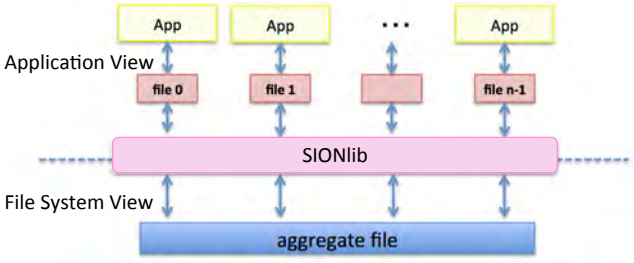


Figure 2: SIONlib

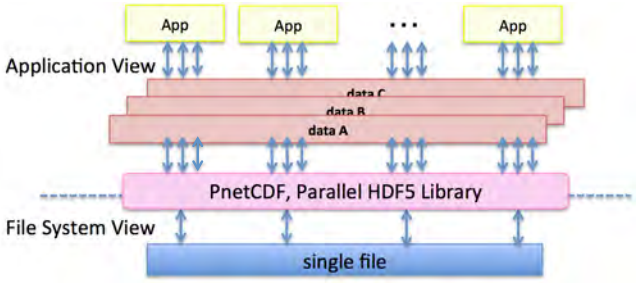


Figure 3: PnetCDF and Parallel HDF5

The remainder of the paper is structured as follows. In Section 2, existing file aggregation techniques are introduced. The concept of the file composition technique is proposed, followed by its design and implementation, in Section 3, and its basic performance is evaluated in Section 4. Section 5 presents a summary.

2. Related Work

Several techniques gather multiple data and store them into one file and that support mechanisms to access the gathered file by parallel processes.

2.1 SIONlib

Figure 2 shows the basic concept of SIONlib [4]. If the I/O pattern of a parallel application is such that each process creates multiple files, N, then the collective SIONlib function must be called N times, and this yields N aggregated files. As a result, the performance gain by the file aggregation can be degraded.

2.2 NetCDF and HDF5

NetCDF [6] and HDF5 [7] are the other I/O aggregation libraries providing self-describing data formats. NetCDF primarily supports a way to access files containing array-oriented data. HDF5 primarily supports a way to access hierarchical data. By using these libraries, the application programmer can describe and store various data into one file with the meta-information about the data and the data format. Parallel NetCDF [3] and Parallel HDF5 [7] are the parallel versions of NetCDF and HDF5, respectively. Both are extended by using MPI-IO and support storing data dispersed among parallel processes (Figure 3).

In both netCDF and HDF5, the data aggregation takes place so that the data structure is preserved. (P)netCDF and (parallel) HDF5 give users a good view of a complex data structure. However, when a user tries to change the file structure, then the aggregated file must be restructured. Thus, users sacrifice the flexibility of their data format.

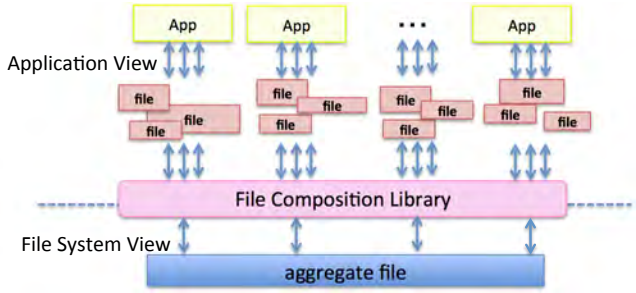


Figure 4: Concept of the file composition technique

3. File Composition Technique

In this section, we propose a new approach to store multiple data generated by a parallel application into a single shared file.

3.1 Proposal

We propose another approach to store multiple data generated by a parallel application into a single shared file by holding an application's I/O pattern. We call the new approach *file composition technique*. Figure 4 shows the basic concept of the file composition technique.

This file composition technique composes data, which looks like a file from the viewpoint of the application, into an aggregated larger file. Unlike SIONlib, (P)netCDF, and (parallel) HDF5, no restrictions are on how the "files" format the application view, and how they are organized and/or structured. Thus, the file composition technique can provide more flexibility than can the existing techniques.

In this paper, we call a file from the application view a "logical file," and a composed file "a physical file." In the application layer, application processes can access various data as if they are individual files. But in the file system layer, all logical files are stored on a single physical file. The file composition library aggregates I/O requests of application processes and translates them into I/O operations for a single shared physical file.

3.2 Design

In this subsection, a design of the implementation of the file composition library is described. We explain how to manage a logical file, how to map logical files to physical files, and how to access the physical file. Figure 5 shows a design overview of the file composition library.

Logical files are managed by our library at the computing node where each logical file has been created, not by a single server such as a metadata server. Even if parallel processes access each individual logical file at the same time, requests are not concentrated at a particular node. Thus, it is expected that the I/O performance does not decrease. Each library process treats logical files as separately divided into a metadata object and data objects. A metadata object holds the information about every logical file, filename, size, position in the physical file, and so on. A data object is an entity of each logical file.

The metadata object and the data objects are split into blocks of a constant size and are distributed into a logical file. By making the block size the same as the stripe size of the file system, each library process can access each individual stripe block without collisions with other processes.

The stripe size of a parallel file system is the maximum message size with which clients and the storage server can communicate. If each library accesses a physical file with the stripe size, then it is expected to have a higher I/O performance. So, each library process has a buffer for data objects with the length equal to the stripe size. If an application process writes many small logical files, then the buffer holds the data of these logical files and writes to a physical file when the buffer becomes full.

4. Evaluation

We implemented the file composition library and evaluated its performance on the Lustre file system. Figure 6 shows the evaluation environment where all of the following experiments took place. The cluster consists of 32 computing nodes and a parallel file system, Lustre. Each computing node has two Intel X5670 CPUs and 96 GiB memory and is connected with the other nodes and the file server nodes by Infiniband 4xQDR. Lustre consists of one Meta Data Server (MDS) and 12 Object Storage Servers (OSSs). Each OSS has 128 Object Storage Targets (OSTs) and the total capacity of the file system is 10 PB. Each OSS has 192 GiB memory. Two switches for the computing nodes and the file system servers are connected with four links. Throughout this evaluation section, the buffer size of the file composition technique is set to 16 MiB.

4.1 Micro Benchmark

We measured the basic performance of our file composition library and the performance of POSIX I/O for comparison. We show the performance of the create, write, and read operations. In each benchmark, the number of processes is increased from one process to 32 processes. Each process runs on one computing node.

On the Lustre file system, files are split into multiple object blocks and each object block is distributed to multiple servers. The size of each object block is called the *stripe size*, and the number of OSTs used to distribute the object blocks is called the *stripe count*. These parameters are factors of the I/O performance. In the case of POSIX I/O, the stripe count is set to 4 and 64, while in the case of using the file composition library, the stripe count is set to 64 and 160. The stripe size is set to 16 MiB constant in all cases.

File Create

Figure 7 shows the time of file creation. Each process repeats the procedure of calling the create and then the close functions for a file 128 times, yielding 128 files. The creations of each file are synchronized and the time is measured from the time to the first create call until the termination of the slowest call of close on the 128th file.

In the case of 32 processes, POSIX with the 4 stripe count took 0.69 seconds and POSIX with the 64 stripe count took 37 seconds. The file composition with the 64 stripe count and with the 160 stripe count took 0.08 seconds.

In all cases, the elapsed time increases as the number of processes increases. But the increasing rate of the file composition case is smaller than that of POSIX. This phenomenon can be considered as that the I/O requests are concentrated on the Meta Data Server in the POSIX I/O. In contrast, the file composition technique can decrease the degree of concentration.

In the POSIX case, when the stripe count gets larger, the creation time gets slower. This is because each create operation involves the operation on the Object Storage Target. In the file composition case, the creation time is constant over the number of processes, independent from the stripe count.

Write

Figure 8 shows the throughput of open-write-close. In this case, each process calls open, write, and close. Each process accesses 128 individual files. The open calls are synchronized and the time is measured from the time to the first create call until the termination of the slowest call of close on the 128th file.

The upper graph of Figure 8 shows the result of the case of setting the file size to 1 MiB, and the lower one shows the result of the case of setting the file size to 16 MiB.

In the case that the file size is set to 1 MiB, the file composition performance is almost twice better than that of POSIX. The assumed reason for this performance improvement is that the file creation time is reduced and that the

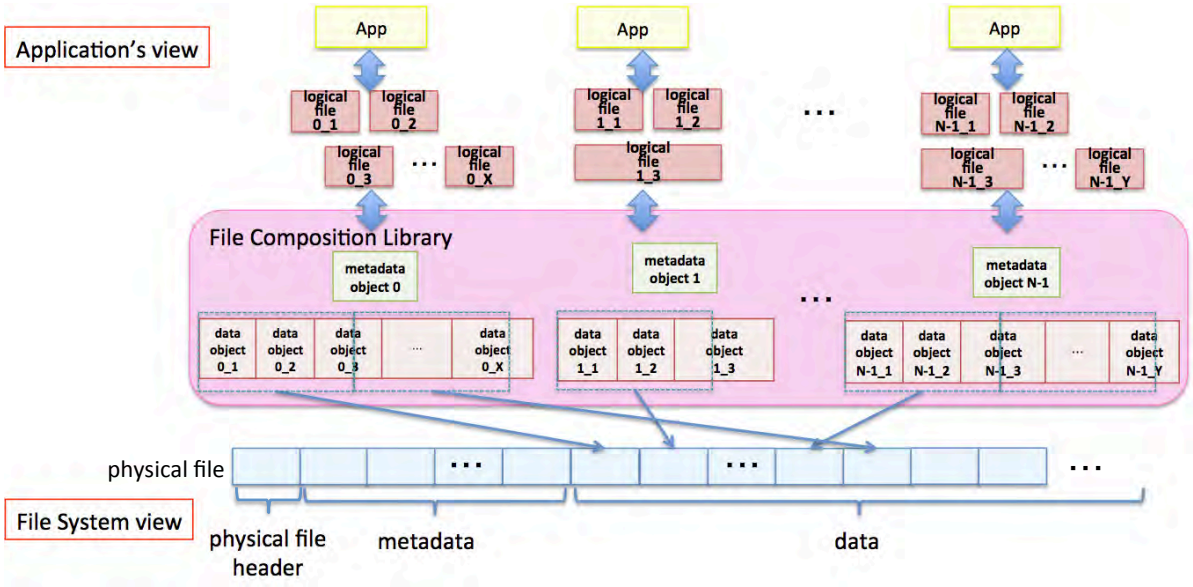


Figure 5: Design overview of the file composition library

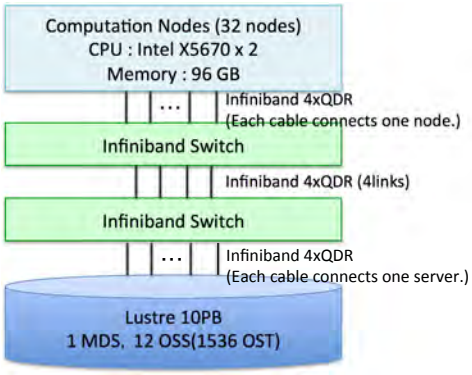


Figure 6: Evaluation environment

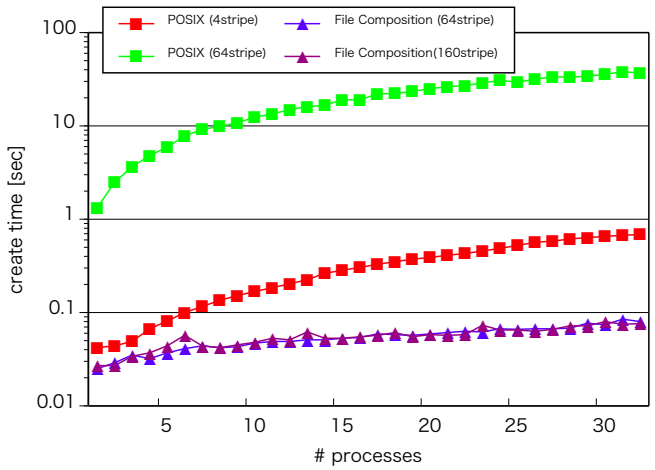


Figure 7: Time of create-close

buffering in the composition technique decreases the number of write calls.

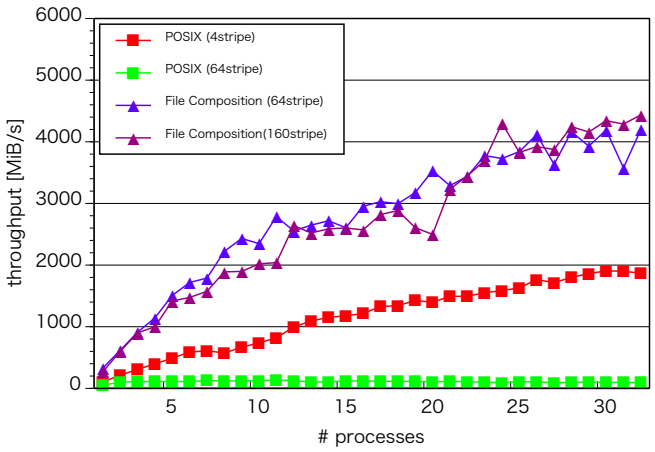
In the case that the file size is set to 16 MiB, the performance of the file composition is approximately twice faster than that of POSIX. This is because the file creation cost is reduced with the file composition technique. However, in the file composition case, when the stripe count is small and the number of processes is large, its performance gain is not high. This is because the number of Object Storage Targets that all processes can access is limited to the stripe counts with the file composition technique.

Read

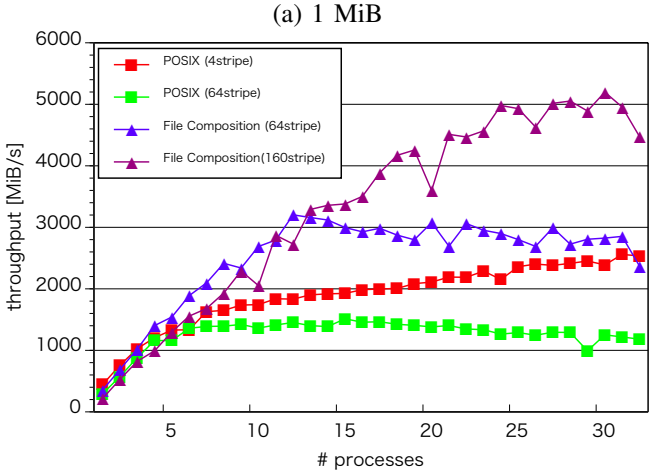
Figure 9 shows the throughput of open-read-close. Each process opens a file and reads data from the file and closes the file. Each process accesses 128 individual files that were created in the previous write benchmark runs. To eliminate the effect of page cache in the Linux kernel, the cache of all computation nodes and all file system servers is cleared before running this benchmark.

The upper graph of Figure 9 shows the result of the case that each file size is set to 1 MiB, and the lower graph shows the result of the case that each file size is set to 16 MiB.

In the case that the file size is set to 1 MiB, the performance of the file composition is approximately three times faster than that of POSIX. In the case that the file size is set to 16 MiB, the performance of the file composition is

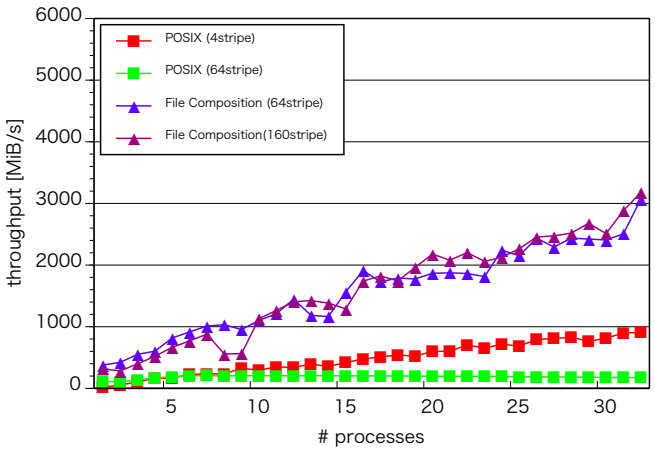


(a) 1 MiB

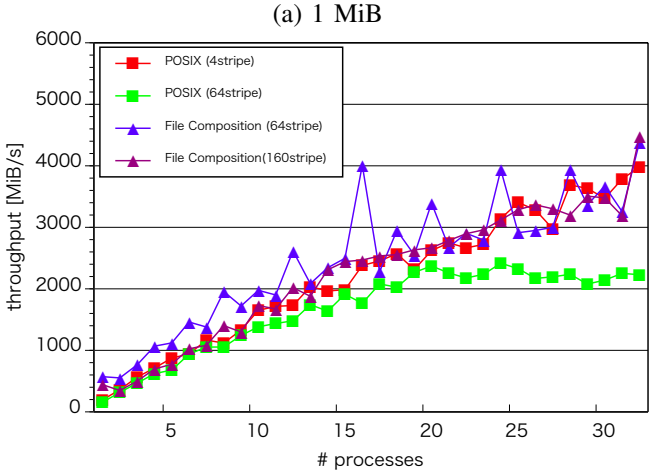


(b) 16 MiB

Figure 8: Throughput of open-write-close



(a) 1 MiB



(b) 16 MiB

Figure 9: Throughput of open-read-close

almost the same as that in the POSIX case. The assumed reason for this performance improvement in the case of 1 MiB is that the process can read the same stripe block as in the case of the file composition. In contrast, in the case that the file size is set to 16 MiB, which is the same size as the buffer of the file composition technique, the file composition technique has no buffering effect.

4.2 Evaluation with a real application

Finally, our file composition technique is applied to a real application that is a climate simulation program, called SCALE, being developed by RIKEN AICS Computational Climate Science Research Team.

When the program is running in parallel, each process periodically calls a file-output function that creates a new file and writes data into the file. When the file-output function is called, first, a new file is created and a file header is written. The file header has meta information about the data to be written in the file. Then, 16 arrays in the program are written and the file is closed.

We rewrote the file-output function by replacing the POSIX I/O functions with the function that our file composition library provides. The file composition library aggregates all files that the SCALE program creates and stores them into a single shared file throughout the whole program execution.

We executed the SCALE program in parallel and measured the elapsed time of the file-output function called by each application process. The program was executed with 32 processes and the size of one array was 10.1 MiB. The SCALE program was set up so that it repeated the file output 10 times. In the POSIX case, the stripe count was set to 0, because that was the fastest parameter variable setting. In the case of the file composition library, the stripe count was set to 160, which is the maximum value we can set.

Table 1 shows the result of the total elapsed time of the output function. The performance of the file composition library was approximately 30% faster than that of POSIX. The assumed reason for this performance improvement is

Table 1: Time of SCALE's file output.

	the fastest process	the slowest process	average of all processes
POSIX	12. 75	15.33	14.43
File Composition	8.79	12.17	10.37

[sec]

that the file composition technique could successfully reduce the number of accesses to a metadata server when each process creates files.

5. Summary

In this paper, we proposed the file composition technique that achieves good I/O scalability of parallel applications without changing the code using the POSIX file I/O interfaces. The library employing this technique aggregates I/O requests of parallel processes and translates them into I/O operations in a single shared file in a parallel file system, which is currently the Lustre file system. Two techniques are integrated to achieve good scalability of I/O operations. The metadata information is separately managed by each process to avoid the contention of metadata operations. The middleware of each process accesses its own stripe block exclusively in the parallel file system to avoid the contention of read/write operations.

The result of the basic performance showed that the proposed library is eight times faster than the regular POSIX I/O library in the case that 32 parallel processes create 128 files each. The results two times better throughput of parallel open-write-close operations and three times better throughput of parallel open-read-close operations. We also applied the file composition technique to the file-output function of a climate simulation program called SCALE. The elapsed time of the file-output function with the file composition library was approximately 30% faster than that of the POSIX I/O. These experiments demonstrated that I/O performance can be improved by using the file composition technique without changing the POSIX file I/O interfaces.

Acknowledgment

This research work was partially supported by JST CREST. We use a cluster that the Information Technology Center at the University of Tokyo provides as a part of the support program for young or female researchers.

References

[1] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," in *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, may 2011, pp. 1 –14.

[2] M. P. I. Forum. Mpi: A message-passing interface standard. [Online]. Available: http://www.mpi-forum.org/docs/docs.html

[3] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netcdf: A high-performance scientific i/o interface," in *Supercomputing, 2003 ACM/IEEE Conference*, nov. 2003, p. 39.

[4] W. Frings, F. Wolf, and V. Petkov, "Scalable massively parallel i/o to task-local files," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 17:1–17:11.

[5] O. Corporation. Lustre file system. [Online]. Available: http://wiki.lustre.org/index.php

[6] U. C. for Atmospheric Research. Netcdf (network common data form). [Online]. Available: http://www.unidata.ucar.edu/software/netcdf/

[7] T. H. Group. The hdf5 home page. [Online]. Available: http://www.hdfgroup.org/HDF5/

Is D the Answer to the One vs. Two Language High Performance Computing Dilemma?

Ralph Butler, Chrisila Pettey, and Matthew Wang
Department of Computer Science, Box 48
Middle Tennessee State University
Murfreesboro, Tennessee 37132, USA

(ralph.butler, chrisila.pettey)@mtsu.edu, mw3n@mtmail.mtsu.edu

Abstract – *During the course of our careers we have written a variety of high performance computing programs (parallel Genetic Algorithms [12, 13], genetic sequence analysis [4], automated reasoning applications such as theorem provers [5], and an asynchronous, dynamic, load-balancing library [1, 3, 7, 10]). In our projects we often found ourselves using a bilingual programming model to gain the advantages of both high performance execution and advanced language features like garbage collected data structures. D gives us access to both facilities in a single language. While D [8] is not a new language in the Computer Science scheme of things, it has only recently advanced to the point where we could begin to consider it as a solution to our dilemma. In this paper we discuss how we use D for rapid development of high performance code, and how we link it to legacy code such as MPICH2 [11] and ADLB.*

Keywords: D Programming, Bilingual Computing, High Performance Computing

1 Introduction

As computer scientists we recognize the value in learning multiple languages. Each language brings a unique set of features to the table that has the potential for providing elegant solutions to various problems. However, in a perfect world, we would only need to be an expert in one language, and that language would have all the features needed to easily create well-designed code. Since the world is not perfect, we have typically in the past focused on one language per project. In practice, the high performance world typically relies on Fortran or C, and in our case it was C. But the lure of scripting languages such as Python would occasionally draw us in. When our project needed the ease of built-in advanced data structures such as associative arrays along with automatic garbage collection of those structures, we used a high level scripting language such as Python. When high performance was absolutely critical or if we were doing low-level programming such as memory management, then we typically used some language that was a C-derivative.

Over time, we began to incorporate both scripting and compiled languages into one project in such a way that distinct pieces of the project might be in different languages, but the interaction between pieces written in different languages was minimal. For example, initially the *mpd* component of *MPICH2* [11] was written in Python while the rest of the modules were written in C [9]. The *mpd* component is a stand-alone process management system that has a trivial interface for connecting to other systems.

Because we always want both performance and ease of development, we have ended up doing bilingual computing. In the bilingual model of computing the interactions between the components written in different languages involve the concept of shared high-level data structures. The reason for two languages was sometimes to overcome a missing feature in C (the high-level data structure) [5]. And sometimes we used two languages to overcome a limiting feature in Python (the global interpreter lock) [6].

However, bilingual computing can be problematic. Besides the obvious requirement of being expert in two languages, bilingual computing introduces the problem of what parts of the project to do in each language and how to interface the various pieces with each other. These are not always trivial problems to overcome. So we are left with the dilemma of do we choose only one language for a project and lose important features of the other language, or do we choose two languages and deal with the problems of interfacing the two languages? And if we choose two languages, someone has to maintain both. As we mentioned previously, MPICH2 was written in C, but its process management component was written in Python. The MPICH2 team, however, felt compelled to rewrite the *mpd* component in C not for performance reasons but largely for ease of maintenance.

The solution to our dilemma is that we need a single programming language that has all the features to elegantly solve our problems. The D programming language [8] was initially developed with the idea of improving C++, and has recently stabilized into what we believe to be the answer to our dilemma. In this paper we begin by discussing salient attributes of D. We then

describe our interfaces to legacy code – specifically MPICH2 [11] and ADLB [1, 3, 7, 10].

2 Why D?

The obvious question to ask is, "Why D?" It doesn't even appear on the TIOBE Programming Community Index [14] of the top 20 most used programming languages. It is interesting to note, however, that the top five programming languages on the list are C derivatives, and four of the remaining languages in the list are scripting languages.

As mentioned before, D was initially developed with the idea of improving C++. Specifically, the developers asked this question:

> Can the power and capability of C++ be extracted, redesigned, and recast into a language that is simple, orthogonal, and practical? Can it all be put into a package that is easy for compiler writers to correctly implement, and which enables compilers to efficiently generate aggressively optimized code? [8]

Because of this, it has many of the features of C/C++ that we consider to be important for high performance computing:

- It is so compatible with C that it will link with existing C programs.
- It has all the features of C from pointers to inline assembly language.
- It has all the features of C++ including a simplified method of handling templates.
- Programming can look like C or C++.
- Performance is equivalent to C.
- The developers claim that on average it compiles 100 times faster than C++ and four times faster than GO. [2]

So we have the performance and features of the C derivative languages. But we also want the high-level data structures, automatic garbage collection, and rapid development time of scripting languages. D actually has many of these features as well. For instance:

- Rapid development can be done with the rdmd wrapper to dmd (the original D compiler) that allows for compiling, linking, and executing without appearing to compile and link.
- D has advanced data structures such as lists and associative arrays.
- D has automatic garbage collection.
- It has Perl-compatible regular expression handling.

In addition to having the aforementioned features of the C derivative languages and the scripting languages, D has some features that can be very helpful for high performance computing projects. Features such as:

- built-in language support for it's own thread model (it should be noted that memory is not shared by default, but it can easily be annotated as shared),
- built-in language support for unit test – both with compiler options and within actual code,
- facilities for contract programming

3 Interfacing D to Legacy HPC Code

Because D is link-friendly with C, i.e., it just links with C functions, one would be inclined to believe that it would also be compile-friendly with C. What we mean by that is you might think that it would allow you to include C header files. But that is not the case. Instead D imports its own modules. For example, you might expect to be able to include the mpi.h header file. Instead, the mpi.h header file has to be converted to a D

```
/* -*- Mode: C; c-basic-offset:4 ; -*-
*/
/*
 *  (C) 2001 by Argonne National
Laboratory.
 *      See COPYRIGHT in top-level
directory.
 */
/*src/include/mpi.h.  Generated from
mpi.h.in by configure.*/
#ifndef MPI_INCLUDED
#define MPI_INCLUDED
/* user include file for MPI programs */
/* Keep C++ compilers from getting
confused */
#if defined(__cplusplus)
extern "C" {
#endif
/* Results of the compare operations. */
#define MPI_IDENT     0
#define MPI_CONGRUENT 1
#define MPI_SIMILAR   2
#define MPI_UNEQUAL   3
typedef int MPI_Datatype;
#define MPI_CHAR
((MPI_Datatype)0x4c000101)
#define MPI_SIGNED_CHAR
((MPI_Datatype)0x4c000118)
#define MPI_UNSIGNED_CHAR
((MPI_Datatype)0x4c000102)
#define MPI_BYTE
((MPI_Datatype)0x4c00010d)
```

Figure 1. A portion of mpi.h.

module. To facilitate this process, there is a *htod* program. *htod* is quite useful in that it accomplishes most of the task, and most of ***what needs to be done by hand*** is annotated in the converted file with comments. Admittedly there is some

hand-crafting necessary to make the project work. The transformation process is shown in Figures 1 – 3 where we list a portion of the mpi.h header file, what that same portion looks like coming out of *htod*, and what the final hand-crafted portion looks like.

```
/* Converted to D from
\mpich2i\include\mpi.h by htod */
module mpi;
/* -*- Mode: C; c-basic-offset:4 ; -*-
*/
/*
 *  (C) 2001 by Argonne National
Laboratory.
 *      See COPYRIGHT in top-level
directory.
 */
/* src/include/mpi.h.  Generated from
mpi.h.in by configure. */
//C     #ifndef MPI_INCLUDED
//C     #define MPI_INCLUDED
/* user include file for MPI programs */
/* Keep C++ compilers from getting
confused */
//C     #if defined(__cplusplus)
//C     extern "C" {
//C     #endif
/* Results of the compare operations. */
//C     #define MPI_IDENT     0
//C     #define MPI_CONGRUENT 1
const MPI_IDENT = 0;
//C     #define MPI_SIMILAR   2
const MPI_CONGRUENT = 1;
//C     #define MPI_UNEQUAL   3
const MPI_SIMILAR = 2;
const MPI_UNEQUAL = 3;
//C     typedef int MPI_Datatype;
extern (C):
alias int MPI_Datatype;
//C  #define MPI_CHAR
((MPI_Datatype)0x4c000101)
//C     #define MPI_SIGNED_CHAR
((MPI_Datatype)0x4c000118)
//C     #define MPI_UNSIGNED_CHAR
((MPI_Datatype)0x4c000102)
//C     #define MPI_BYTE
((MPI_Datatype)0x4c00010d)
```

Figure 2. A portion of mpi.h that has been run through *htod*

```
/* Converted to D from
\mpich2i\include\mpi.h by htod */
module mpi;
/* -*- Mode: C; c-basic-offset:4 ; -*-
*/
/*
 *  (C) 2001 by Argonne National
Laboratory.
 *      See COPYRIGHT in top-level
directory.
 */
/* src/include/mpi.h.  Generated from
mpi.h.in by configure. */
//C     #ifndef MPI_INCLUDED
//C     #define MPI_INCLUDED
/* user include file for MPI programs */
/* Keep C++ compilers from getting
confused */
//C     #if defined(__cplusplus)
//C     extern "C" {
//C     #endif
/* Results of the compare operations. */
//C     #define MPI_IDENT     0
//C     #define MPI_CONGRUENT 1
//C     #define MPI_SIMILAR   2
//C     #define MPI_UNEQUAL   3
const MPI_IDENT = 0;
const MPI_CONGRUENT = 1;
const MPI_SIMILAR = 2;
const MPI_UNEQUAL = 3;
//C     typedef int MPI_Datatype;
extern (C):
alias int MPI_Datatype;
//C     #define MPI_CHAR
((MPI_Datatype)0x4c000101)
//C     #define MPI_SIGNED_CHAR
((MPI_Datatype)0x4c000118)
//C     #define MPI_UNSIGNED_CHAR
((MPI_Datatype)0x4c000102)
//C     #define MPI_BYTE
((MPI_Datatype)0x4c00010d)
const MPI_CHAR =
cast(MPI_Datatype)0x4c000101;
const MPI_SIGNED_CHAR =
          cast(MPI_Datatype)0x4c000118;
const MPI_UNSIGNED_CHAR =
          cast(MPI_Datatype)0x4c000102;
const MPI_BYTE =
cast(MPI_Datatype)0x4c00010d;
```

Figure 3. A portion of mpi.h that has been run through *htod* and then hand-crafted to work correctly

While the conversion process was a bit tedious, it was not difficult, and we were successful in using a small

subset of MPICH2 in D programs. We turned our results over to the current MPICH2 support team, and discussions are being held to determine if MPICH2 is going to support D as it already does C, C++, and Fortran.

Having this experience behind us, it was relatively trivial to perform the same operation on ADLB and begin using ADLB in D programs. Encouraged by these results, this semester we have provided D as an option in our graduate level parallel processing class which uses both MPI and ADLB.

As a purely intellectual exercise to convince ourselves that D would be a good replacement for C and/or Python, we did the necessary work to convert portions of mpd, ADLB, and the theorem prover into D. In cases where our major concern was nothing but performance, we found that writing D code was essentially equivalent to writing C code. If we wanted to, we could use pointers and write memory managers. On the other hand, if we wanted to let D handle memory management we could. In fact, D was able to handle practically all that we wanted it to handle. There were only two places where D was somewhat less than perfect for our needs. The first tiny flaw is that D's definition of associative arrays is not as elegant as Python's. If you want to map a single data type to another single data type, D is fine. But if you want to use multiple data types as the key, you have to use a *Variant* – so technically D is up to the task but it is not as elegant as Python. The second tiny flaw is that as far as we can tell, there is no serialization library right now in D. There is one proposed that has not been accepted. But to fully re-implement some of the mpd code, we would need serialization. As part of our exercise, we implemented a small serialization library suitable for our own needs, but it would be nice to have serialization as part of the distribution libraries.

4 Conclusions and Future Work

During the course of several decades of writing high performance code our goals have evolved. Initially the aim was simply to improve performance – which meant programming in C (or some C derivative). Over time our goals evolved to allow more features of scripting languages for small parts of a large project – features such as advanced data structures and automatic garbage collection. Eventually our goals would require the sharing of data structures between high-level scripting code and low-level performance code in a single project. The desire for both caused us to develop a bilingual programming model, thus requiring us to deal with the problems associated with sharing data structures between languages. This was not an ideal situation.

We believe the dilemma of whether to code a project in a single language or two languages can be solved by coding in D. D provides the power and capability of the C derivative languages. It also provides the flexibility of the scripting languages. And D has some added attributes that aren't available in the others. It is trivial to link D code to C code, and it is fairly easy to interface D to legacy code using the *htod* software. The interface to legacy code does require some manual labor to get the project to work, but it is minimal.

We have begun using D in our graduate Parallel Processing class as well as in our research. The elegance of the D threading model and its ease of use with MPI and ADLB have led to plans to use D next year in a Software Design and Development course which might not otherwise be able to use high performance computing facilities.

While we can use D on our own cluster, on machines like the BlueGene/Q, this might not currently be possible. Typically you can only be guaranteed C/C++ and Fortran. Since D is under consideration by the MPICH2 team, this may lead to availability of D on the bigger clusters/machines.

Just as people constantly argue about what language is better. They also have frequent discussions about whether or not languages are scalable. For example, they might say, "Perl's fine for quick and dirty hacks, but it's not scalable." Or they might say, "Python is elegant and scalable." We doubt that anyone would argue that C/C++ is not scalable. On the other hand, the large projects done in C/C++ may not be as elegant or as maintainable as those done in Python. All arguments aside, if you want performance, rapid development, elegance, scalability, safe language features, ease of maintenance, and advanced software engineering functionality, then it is really hard to find it all in one place other than D. At this point we see no reason to use anything other than D for new projects. From our point of view it scales elegantly, and gives us all the features we need.

5 References

[1] ADLB: Asynchronous Dynamic Load Balancing, http://www.cs.mtsu.edu/~rbutler/adlb Accessed March 2012.

[2] Alexandrescu, A., "Three Cool Things About D – The Case for the D Programming Language," Google Tech Talk, July 29, 2010, http://www.youtube.com/watch?v=RlVpPstLPEc Accessed March 2012.

[3] ASCR SciDAC Universal Nuclear Energy Density Functional project: A Closer Look at Nuclei, Building a Universal Nuclear Energy Density Functional. http://www.scidac.gov/physics/unedf.html Accessed March 2012.

[4] Butler, R., Butler, T., Foster, I. Karonis, N., Olson, R., Overbeek, R., Pfluger, N., Price, M., and Tuecke, S., "Aligning Genetic Sequences," Chapter 11 in Strand: New

Concepts in Parallel Programming, Foster and Taylor, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[5] Butler, R., and Pettey, C., "A Bilingual Theorem Prover for Evaluating HPC Systems," Proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, June 2007, pp 1000 – 1003.

[6] Butler, R., Ells, D., and Pettey, C., "PySMO: Python Shared Memory Objects," Proceedings of the 2010 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, July 2010, pp 203 – 209.

[7] Butler, R., Pettey, C., and Manifold, B., "Go2ADLB: An Interface for Using ADLB Within Go," Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, July 2011, pp 54 – 58.

[8] D Programming language http://dlang.org/index.html Accessed March 2012.

[9] Gropp, W., Lusk, E., Ashton, D., Balaji, P., Buntinas, D., Butler, R., Chan, A., Krishna, J., Mercier, G., Ross, R., Thakur, R., and Toonen, B., MPICH2 Installer's Guide, September 14, 2007, ftp://ftp.mcs.anl.gov/pub/mpi/mpich2-doc-install.pdf, Accessed March 2012.

[10] Lusk, Ewing L., Pieper, Steven C., Butler, Ralph M., "More Scalability, Less Pain," SciDAC Review 2010 http://www.scidacreview.org/1002/html/adlb.html Accessed March 2012.

[11] MPICH2, www.mcs.anl.gov/research/projects/mpich2/ Accessed March 2012.

[12] Pettey, C. adn Leuze, M., "A Theoretical Investigation of a Parallel Genetic Algorithm," Proceedings of the Third International Conference on Genetic Algorithms, 1989.

[13] Pettey, C., An Analysis of a Parallel Genetic Algorithm, PhD dissertation, Vanderbilt University, May, 1990.

[14] Tiobe Programming Community Index for February 2012 http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html Accessed March 2012.

Minimum-Blocking Parallel Bidirectional Search

Dale E. Parson
Kutztown University of PA
15200 Kutztown Road
Kutztown, PA 19530-0730

Dylan Schwesinger
Lehigh University
Memorial Drive West
Bethlehem, PA 18015

Abstract

The present work investigates using non-blocking and minimum-blocking Java library classes as a basis for improving performance of parallel bidirectional search on a multiple-instruction multiple-data (MIMD) processor. The approach represents individual states as minimum-size, immutable objects. It uses a work queue to distribute states-for-expansion among worker threads, and it uses two sets for keeping track of states previously explored in each direction. The queue class is thread-safe and non-blocking, and the set class is thread-safe and non-blocking for read operations, with parallel locking of subsets for write operations. It is essentially a dataflow approach as opposed to a state machine approach. Rather than step worker threads through state transitions using blocking synchronization, it flows states to be expanded to worker threads in the order required by bidirectional search. This approach has clear, measurable advantages over approaches that use blocking synchronization.

Keywords

bidirectional search, concurrent programming, Java, multiprocessing, parallel programming

1. Introduction and related work

This report is an outcome of curriculum development for a senior and graduate-level course in multiprocessor programming.[1] The course uses the Java™ programming language because of its extensive library of thread-safe container classes and atomic data types, and its explicit memory model that supports aggressive optimization of dynamically compiled code [1]. We have found that for some algorithm benchmarks Java outperforms statically optimized C/C++ using the native compiler. This report focuses on applying Java library classes to the problems of parallel bidirectional search.

Bidirectional search is a classic approach to solving search space problems when both the initial and final states of the search are known in advance [2]. It searches for paths that connect these two states, typically searching for minimum-length paths. In problems with exponential growth of the search space size as a function of search path length, bidirectional search reduces the number of states inspected over unidirectional approaches by integrating the results of two shorter paths that grow simultaneously from the initial and final states.

Bidirectional search is an interesting algorithm for adaptation to parallel programming because it aims at improving run-time performance over simpler search algorithms such as depth-first or breadth-first search, and because it lends itself to parallel implementation. Figure 1 is a schematic view of bidirectional search as exploration of a maze in search of the shortest path, given knowledge of both the entrance and exit locations. Regardless of the concrete problem being solved, bidirectional search always requires knowledge of the starting and ending states of the search. It often utilizes problem-specific heuristics to prune the search space of dead ends, but it is not required to do so.

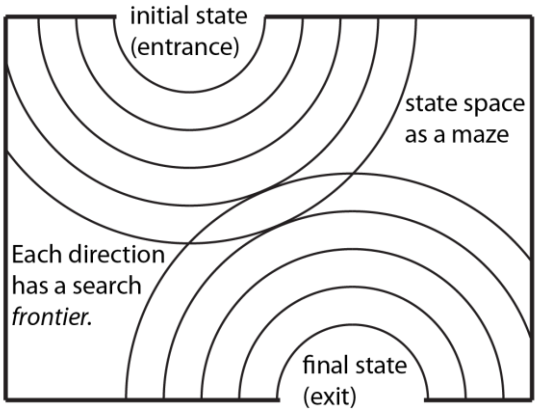


Figure 1: Bidirectional Search as a Maze

The fundamental point of bidirectional search is to limit the exponential growth in number of states explored in a single direction by exploring two shorter paths, one from each direction, and then detecting states in which those opposing paths meet. The outermost set of states currently being explored in either direction constitutes that direction's *frontier*. A single-threaded search based on breadth-first search uses a first-in first-out (FIFO) queue of states to expand as a work queue. The algorithm first enqueues the initial state and final state in the work queue, after which it iteratively removes a state, computes its single-step expansions, checks for cycles and converging DAG paths within the states of its

[1] This work was made possible by equipment grants from Sun Microsystems and the NVIDIA Corporation, and by stipend grants from the Intel Corporation and the PA State System of Higher Education. Please see the Acknowledgements section for details.

originating direction, and checks for collisions with states coming from the opposite direction. Cycle / converging paths and collision checking require storing explored states in a set (keyed on location in the space + search direction) or two sets (keyed on location only). Detection of opposing-path collisions uncovers shortest-path solutions to the problems. In the absence of cycles / converging paths and solutions, the algorithm enqueues one or more single-step expansions and repeats these steps until it locates a solution.

The worst case time, space complexity for unidirectional breadth-first search is $\mathbf{O(b^{d+1})}$, where base **b** is the number of alterative branches (*branching factor*) in the search path that can be taken at any step, and exponent **d** is the *depth* (or equivalently *length*) of the path. When b==3, for example, an un-pruned frontier contains 3 possible states after 1 step, 9 possible states after 2 steps, and so on, generalizing to $\mathbf{b^d}$ states at the frontier, although some may be eliminated through detection of cycles, converging DAG paths, or via application-specific heuristics. The total states explored leading up to the frontier + the frontier itself grows at the rate $\mathbf{O(b^{d+1})}$.

Bidirectional breadth-first search, in contrast, grows at the much lower rate $\mathbf{O(b^{d/2})}$. Each of the two search directions in bidirectional search grows to only half the length of the corresponding unidirectional search, thereby cutting down on the massive exponential growth in explored states that comes with the relative doubling of length in unidirectional search.

Recent work reported on integrating parallel processing with bidirectional search focuses on applying parallel implementation of heuristic strategies to prune the search space [3-5]. Using application-oriented heuristics to radically reduce the number of states explored is the primary means for accelerating the basic bidirectional algorithm. Observing the incremental state expansion of a search domain often uncovers useful heuristics.

2. Minimum Blocking Approach

Our initial solution to parallel bidirectional search used the following algorithm, which implements a two-phase state machine. Each immutable state object contains its internal state fields and an immutable reference to its predecessor in its search path.

```
enqueue initial state into work queue
enqueue final state into work queue
set forwardStatesSet to the set of {initial state}
set backwardStatesSet to the set of {final state}
set setOfSolutions to empty set {}
set direction to Forward
set isdone flag to False
while not isdone
    dequeue a state-to-expand from front of work queue
    if directionOf(state-to-expand) not equal direction
        post a pending-change-of-direction to all threads.
        block until all threads ready to reverse direction.
        set direction to its reverse.
    for each single-step expansion of state-to-expand
        if expansion is in StatesSet from opposing side
            if 1st solution or cost equals solution's cost
                add expansion's path to setOfSolutions
            else (cost is greater)
                set isdone flag to True
        else if expansion is in StatesSet from this side
            // a cycle or converging DAG path detected
            do not use this expansion
        else
            add expansion to StatesSet from this side
            enqueue expansion in work queue
```

Listing 1: Parallel, blocking state machine

Enqueues into the work queue and dequeues from the work queue do not block in this algorithm. The viability of non-blocking retrieval depends on the fact that exponential growth of the search space ensures that most dequeue operations will receive a state-to-expand from the work queue. It is only at the beginning of the search that some threads do not initially find states-to-expand via the non-blocking dequeue operation within a given phase (forward or backward). Those threads resort to a polling loop, trying the queue repeatedly until they receive data or until another thread posts a *pending-change-of-direction* flag. Idle polling consumes processors only until the work queue begins to grow at an exponential rate. Our implementation uses the ConcurrentLinkedQueue from the java.util.concurrent library package as the work queue. The documentation for that class states that, "This implementation employs an efficient "wait-free" algorithm." [6, 7]

The *forwardStatesSet* and *backwardStatesSet* of Listing 1 are objects of class ConcurrentHashMap of java.util.concurrent. There is no comparable Set class per se, but the keys of a Map can serve as elements of a Set. The documentation for this class states, "However, even though all operations are thread-safe, retrieval operations do *not* entail locking." Write locks are distributed across a number of *stripes*, where a stripe is a subset of the buckets in the hash table [8]. When two writers do not collide on the same stripe, they do not impede each other. Application programmers can adjust the number of stripes, trading increased parallelism against the memory cost of maintaining additional lock stripes.

A change of direction in the algorithm of Listing 1 entails waiting until all threads have completed expansion of the current direction, forward or backward. Referring to Figure 1, all threads expand the frontier of only one direction at a time during one phase of this state machine approach. Our thinking was to keep the set of states coming from the opposing side stable for collision testing during the expansion of states in the current side. We implemented blocking using the CyclicBarrier class from java.util.concurrent [6]. This library class blocks all calling worker threads until the last worker thread has entered the barrier. The final thread to enter reverses the direction of the search state variables, and then all threads enter the next phase of the search. CyclicBarrier provides very coarse-grain synchronization. The intent in

using CyclicBarrier was to minimize fine-grain synchronization while supporting stability in testing for state membership in the opposing *StateSet*.

After working with our implementation of Listing 1 we realized that we could eliminate locking altogether. Listing 2 gives the revised algorithm.

```
enqueue initial state into work queue
enqueue final state into work queue
set forwardStatesSet to the set of {initial state}
set backwardStatesSet to the set of {final state}
set setOfSolutions to empty set {}
set isdone flag to False
while not isdone
    dequeue a state-to-expand from front of work queue
    for each single-step expansion of state-to-expand
        if expansion is in StatesSet from opposing side
            if 1st solution or cost equals solutions' cost
                add expansion's path to setOfSolutions
            else (cost is greater)
                set isdone flag to True
        else if expansion is in StatesSet from this side
            // a cycle or converging DAG path detected
            do not use this expansion
        else
            add expansion to StatesSet from this side
            enqueue expansion in work queue
```

Listing 2: Parallel, minimum-blocking dataflow machine

The dataflow algorithm of Listing 2 dispenses with the CyclicBarrier waiting of Listing 1. The dequeue operation remains a non-blocking poll with looping until the work queue begins to fill, tests for *StatesSet* membership are non-blocking, and insertion of new states into *StatesSet* occurs using concurrent lock stripes. With a high lookup-to-insertion ratio for *StatesSet* members (all insertions are preceded by lookups to detect cycles and solutions), locking is minimal and configurable via the *StatesSet's* stripes constructor parameter.

Dispensing with coarse-grain synchronization of the two-phase state machine is possible because states flow through the *work queue* in approximately the correct order. Forward states-to-expand alternate with reverse states-to-expand, partitioned by frontier-being-expanded for the most part.

This temporal sequencing of wave fronts is stochastic, not deterministic. A thread that finds most (but not all) of its state expansions to be dead ends (cycles or converging DAG paths) for a series of dequeue operations places frontier states onto the work queue quickly; some worker threads could be two phases ahead of other threads, expanding a path of length L+1 for a given direction while some threads are expanding paths of length L for that same direction. There is no particular problem in occasionally "getting ahead," as implied by the overlapping frontiers of Figure 1. A thread that has gotten ahead on one turn may find an opposing path one level deeper into the opposing side's search space, but the discovered path is still a solution path. The algorithm retains only the set of minimum-length solution paths. Normally, by the time a thread has reached level N+1 in the search from its dequeued state-to-expand's origin, all other threads have dequeued all level N states from the work queue, and they will complete expansion of those N-level states before checking the *isdone* flag set by the first solution's discovery. Some of those N level expansions may be redundant with the N+1 level solution from the thread that "got ahead." The algorithm discards such redundant solutions.

There is still a potential problem, although we have not seen it in practice. The algorithm of Listing 2 makes it possible for some advanced state-to-expand to be multiple frontier levels ahead of other states being expanded in the same direction. If the thread that is expanding a level N+2 (or higher) state sets the *isdone* flag while other level N states are being expanded in the same direction, then some solutions could be missed in an exhaustive search for all distinct minimum-length paths. The fix is to discard a state-to-expand after a solution has been found, if the state-to-expand has a path length greater than the integer ceiling of ½ of the known solution's length, setting the *isdone* flag at that point. The length of the first known solution helps to prune state expansions. At the point that the work queue becomes empty after the *isdone* flag is set to True, worker threads can terminate their work. Of course, a thread may detect this condition and terminate just before another thread enqueues a state-to-expand, but at that point processing is converging on the last of the solutions, and the thread that enqueued the state-to-expand is guaranteed to be available to dequeue that state-to-expand, if no other thread gets there first. States-to-expand in possible solution paths will not be left in the work queue by all terminating threads, and detection of the *isdone* flag in combination with an empty work queue indicates convergence on the last of the solution paths.

One final problem with the fact that some states-to-expand may get multiple levels ahead of most states being expanded in a direction is the fact that these states may not lie in a solution path. The advanced state has gotten outside of the intersecting cones of the frontiers of Figure 1. This problem is a small efficiency concern, not a bug. There may be some unnecessary searching outside the intersecting frontiers of Figure 1, but the performance impact is insignificant compared to the benefits of the minimum-blocking algorithm. Exploration of such states does not lead to false solutions or premature termination.

3. Shortest path performance

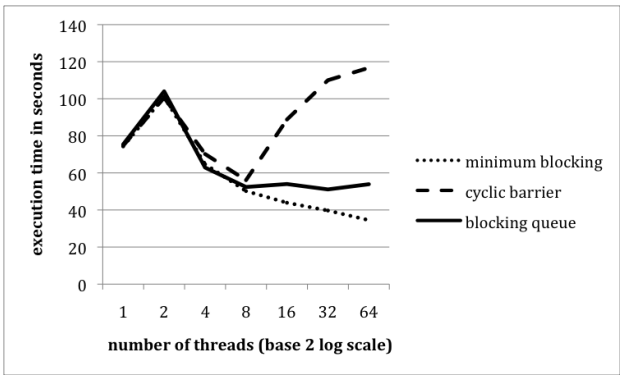
For the performance measurements of this section we ran two representative applications of bidirectional search. The first finds the solution of the so-called "Penny-Dime problem," where there is an arrangement of some number N of pennies P, followed by one blank space, followed the same number N of dimes D. The goal is to find the series of moves that will reverse a sequence such as

PPPPPP_DDDDDD to the sequence DDDDDD_PPPPPP. Legal moves consist of moving a coin one location into the space, or jumping a coin over a single neighbor (as in checkers) to the space. Heuristics such as avoiding retrograde moves can accelerate the search, but the overall form of the algorithm remains unchanged.

The other benchmark, which is the one reported here, is a simplification of a maze construction problem. The original algorithm searches for non-shortest paths that match some minimum-length threshold, in the interest of constructing interesting mazes.

For the benchmark reported here, we use an algorithm that simply searches for the shortest path from the maze entrance to its exit by crossing empty space with no obstacles other than outside walls. When the program starts, there is a pseudo-randomly selected entrance location, exit location, and outer walls, but the inside of the proposed maze is empty at that point. The algorithm simply finds the shortest path between two points, where the search from the entrance does not have knowledge of the location of the exit, and the search from the exit does not have knowledge of the location of the entrance. This is essentially blind search.

The machine used for this benchmark is a 64-threaded Sparc server obtained via a 2009 grant from Sun Microsystems [9]. It houses 8 cores x 8 threads-per-core = 64 hardware threads, 16 Gbytes of main memory, and a 1.2 Ghz clock speed. Each core is limited to a rather small 16 Kbytes of L1 instruction cache and 8 Kbytes of L1 data cache, with 4 Mbytes of L2 cache distributed across the cores. While not the best fit for large data sets with random access patterns due to the limited cache, it performs admirably when running Java programs with good memory locality or modest memory consumption. The individual states of the bidirectional search problems that we have benchmarked are small, although references to a large number of these small state objects can reside in the work queue and sets of the algorithm.

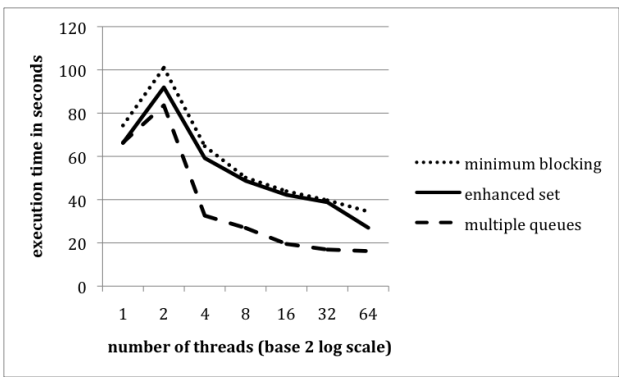


Graph 1: Multithreaded bidirectional maze construction

Graph 1 plots execution time in real seconds on the Y axis as a function of number of threads on the logarithmic X axis for construction of a minimal path of length 2947 in a 2500 x 2500 space using blind bidirectional search. The dashed *cyclic barrier* curve of Listing 1 reaches its peak performance at 8 threads (56.1 seconds compared to 74.7 seconds of its single-threaded case), after which execution time grows with the number of hardware threads employed. Normally adding threads beyond some optimal spot for an algorithm increases start-up and scheduling overhead, although in this case scheduling overhead is minimal because all 64 hardware threads are available for execution. The problem here is that when one or two threads lag behind the others in the exploration of a frontier, the remaining 30-to-31 or 62-to-63 threads block idly in the cyclic barrier until those one or two threads enter the barrier. With only 8 hardware threads employed, each thread has more states to expand for a given frontier, and the cost of exploration averages more evenly across the threads, so there are fewer opportunities for entering this degenerate state repeatedly. More threads wait repeatedly until time-consuming laggards complete their work in the 64-threaded case. In the 8-threaded case the per-thread workload averages out, minimizing the stalling effect of the cyclic barrier.

The dotted *minimum blocking* curve of Listing 2 starts at 74.3 seconds, shows a loss of performance due to two-threaded contention at 101 seconds, after which execution time decreases consistently to 34.5 seconds for 64 threads.

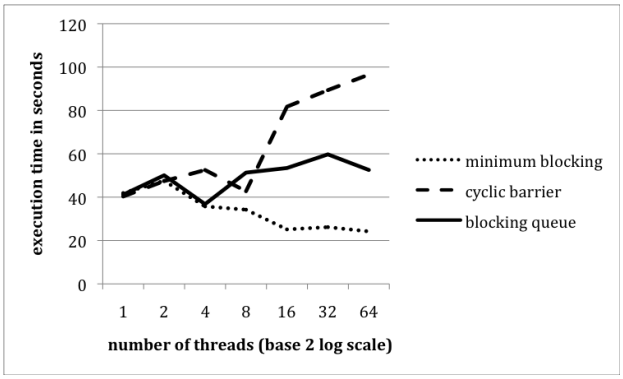
The final, *blocking queue* curve shows the result of replacing the ConcurrentLinkedQueue of *minimum blocking* case with the LinkedBlockingQueue class of the Java library, and using the blocking take() method instead of the non-blocking poll() method for dequeueing states-to-expand. Replacing queue polling with blocked waiting increases processing and scheduling overhead by the Java Virtual Machine and operating system. Polling, even when its finds no work in the queue, is better for this application because it avoids calls into the operating system. Given the exponential growth in number of states to be searched, most dequeue calls for LinkedBlockQueue do not block, yet the cost of calling dequeue methods that must acquire and release locks is clear from looking at Graph 1. After starting at 75.4 seconds for the single-threaded case and then rising to 104 seconds for the double-threaded case, the blocking queue approach drops to 52.4 seconds at 8 threads and then essentially levels off.



Graph 2: Improved nonblocking data structures

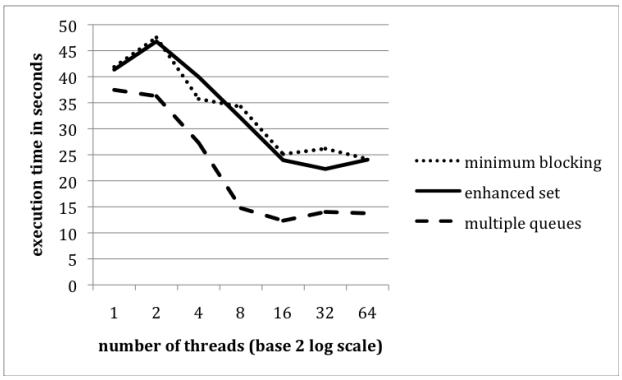
Because the overall halving of execution time in going from 1 thread to 64 threads for the *minimum blocking* approach is disappointing, we decided to attempt to tune the thread-safe data structures to get additional gains. Graph 2 shows the results. The dotted *minimum blocking* curve is the same as in Graph 1. The solid *enhanced set* curve shows the modest gains resulting from initializing the *StatesSet* implemented using ConcurrentHashMap to its known maximum size (4 million elements, determined empirically), reducing its load factor from the default .75 to .5, and increasing its number of lock stripes from the default 16 to 128. Initializing the set size reduces repeated growth overhead. Reducing the load factor reduces hash table collision overhead, and increasing the number of lock stripes reduces contention among concurrent writing threads.

More substantial gains come with the *multiple queues* test case that allocates one work queue per worker thread. Whenever a worker thread is about to enqueue a new state-to-expand, it increments an atomic integer and uses that value as an index to a thread-specific queue. The round-robin nature of enqueuing reduces the probability of thread contention for enqueuing, because a given queue will have its enqueue operation invoked only once for every T enqueue operations, where T is the number of worker threads. A given queue will have dequeue invoked only by its worker thread, eliminating dequeue contention entirely. This *multiple queues* approach bottoms out at an execution time of 16.2 seconds for the 64-threaded case, as compared with 27 seconds of the *enhanced set* approach and the 34.5 seconds of the basic *minimum blocking* approach at 64 threads. The multiple queue approach basically yields a second doubling of performance from its 66.4 second starting point when compared to the other approaches of Graph 2.



Graph 3: Graph 1 benchmarks on a 16-threaded Opteron

Graphs 3 and 4 repeat the benchmarks of Graphs 1 and 2 on a 16-threaded AMD Opteron server also obtained via a grant from Sun Microsystems. The server houses 8 cores x 2 threads-per-core = 16 hardware threads, 32 Gbytes of main memory, and a 2.7 Ghz clock speed. Each core has a substantial 128 Kbyte L1 cache and a 1 Mbyte L2 cache.



Graph 4: Graph 2 benchmarks on a 16-threaded Opteron

Differences in machine architectures such as instruction sets or cache sizes can make substantial differences in performance curves, but in this case the curves of Graphs 3 and 4 repeat the dynamics of Graphs 1 and 2.

4. Conclusions and future work

Bidirectional search is amenable to minimum blocking implementation using immutable state objects, non-blocking queuing of states-to-expand, non-blocking set membership tests in checking for cycles, converging DAG paths and solutions, and parallel lock stripes for updating sets. Java's ConcurrentLinkedQueue and ConcurrentHashMap library classes are excellent matches for the queue and set data structures required by this approach. Adjusting initial set size, load factor and lock striping can contribute modest performance enhancement. Replacing a single non-blocking work queue with multiple work queues that are written by all threads in round-robin order, and that are read respectively by only a single worker thread, leads to a second doubling of performance over the basic minimum blocking approach.

We anticipate porting this work to a NVIDIA Tesla graphical processing unit (GPU) in search of further performance gains. Initial study indicates that a heterogeneous MIMD CPU / GPU approach may be effective. A multithreaded CPU can construct multiple queues for GPU processing elements, one per element, during a CPU phase, along with building the required sets in GPU global, read-only memory. During a GPU phase, the processing elements drain their respective queues, discard cyclic and converging-path states, and send queue and set updates back to the CPU phase. The CPU can update set membership in global, read-only memory incrementally, redistribute the queues, and resume the GPU phase for the next round of state expansion. There are many details remaining to be ironed out in this basic plan.

5. Acknowledgements

This work was made possible by the generous grant of three multiprocessor servers from Sun Microsystems in 2009. A curriculum development grant from the PA State

System of Higher Education supported the initial round of designing benchmarks and course projects such as the one described here. A second curriculum development grant from Intel's *Parallelism in the Classroom* program is making ongoing extension of such materials possible. Finally, an equipment grant from NVIDIA makes the exploration of a Tesla GPU implementation of this work possible.

6. References

[1] Goetz, Brian, et. al., *Java Concurrency in Practice*, Addison-Wesley, 2006.

[2] I. Pohl, "Bi-Directional Search", Machine Intelligence, 1971, pp. 127-140.

[3] A. Toptsis, R. A. Chaturvedi, and A. Feroze, "Kohonen-guided Parallel Bidirectional Voronoi-assisted Heuristic Search," *International Journal of Advanced Science and Technology* Vol. 5, April, 2009.

[4] P.C. Nelson, "Parallel Bidirectional Search Using Multi-Dimensional Heuristics", Ph.D. Dissertation, Northwestern University, Evanston, Illinois, June 1998

[5] P.C. Nelson, and A.A. Toptsis, "Unidirectional and Bidirectional Search Algorithms", *IEEE Software*, Vol. 9, No. 2, March 1992, pp. 77-83.

[6] Oracle Corporation, documentation for classes in package javat.util.concurrent, http://docs.oracle.com/javase/6/docs/api/index.html, March 2012.

[7] M. M. Michael and M. L. Scott, "Simple, Fast and Practical Non-blocking and Blocking Concurrent Queue Algorithms," *Proceedings of Fifteenth ACM Symposium on Principles of Distributed Computing* (PODC '96), Philadelphia, PA, 1996.

[8] Herlihy and Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

[9] Fujitsu, Sparc Enterprise T5120, T5220, T5140 and T5240 Server Architecture, http://www.fujitsu.com/downloads/SPARCE/whitepapers/T5x20-T5x40-wp-e-200907.pdf, July 2009.

Concurrency Control and Recovery of Long Lived Transaction Processing in Virtualized Environment

Nazifa Noor, Motoyasu Nagata
Department of computer science, Osaka Kyoiku University, Osaka, Japan

Abstract *- This paper explores long lived transaction processing in the virtualized environment. The employment of the virtualization is motivated by the information sharing among database servers, which enhances concurrency of long lived transaction processing system. Our long lived transaction processing system is characterized by the combination of concurrency control and recovery. Our protocol exploits priority ceiling instead of traditional locking method, while employing and extending the concept of 'wait' inherent to altruistic locking approach. The recovery is efficient due to information sharing log, active list, commit list and abort list. Also, we have evaluated the performance of our approach and shown its effectiveness.*

Keywords: Long Lived Transaction Processing, Priority Ceiling, Two-phase-locking, Virtual Environment, Concurrency Control, Recovery techniques.

1 Introduction

As the database technologies are adapted to a wide range of applications, long lived transaction processing models are inevitably required to support various semantics associated with the applications for throughput enhancement. The ability to maintain concurrency control, and recover from erroneous execution or failure is essential requirement for long lived transaction processing system.

Long Lived Transactions (LLT), have long duration compared to the other transactions, therefore these transactions cause significant delays for Short transactions in the database systems, this compromises concurrency of the system.

In this paper, our goal is developing protocols to motivate concurrency of those database systems that contain long lived transactions. Our approach is exploiting priority ceiling to the long lived transactions in a virtual environment. Furthermore enhancement in the properties of altruistic locking is proposed. We claim that our proposed method brings more concurrency to transaction processing system. The concurrency control is the activity of coordinating concurrent accesses to database in multi-user database system. Concurrency in virtual environment permits users to access a database in a multi-programmed fashion with the illusion that access to databases of various location and servers is possible from one and nearest server. The concurrency control has been actively investigated several times for the past decade. One standard solution accepted to cope with the problem is *Two-Phase-Locking* method, which is applicable to any type of transaction processing system.

Considering the fact that long lived transactions (LLTs) conduct lengthy computation over database objects and their process interval is longer than others, applying two-phase-locking sounds too rigid.

By two-phase-locking method transactions encountering lock conflict are blocked for long period of time, which slow down the processing system. Basic Altruistic locking is a pioneering attempt to cope with concurrency control of LLTs; this concept sets the term of donating database objects (release before unlock). Database objects which are no longer needed by the transaction are released during the process interval of that transaction before it's terminated. Using altruistic locking, short lived transactions (SLT) can go into the state called *wake of LLTs* which means they can access the database objects donated by LLT.

2 Proposed Approach

Database servers from distinct and remote locations are virtually centralized in a way that gives user the illusion as they are located together in one and closest server. In virtual environments transaction manager TM, scheduler and database manager DM are centralize and share. This makes long lived transactions processing smoother than the distributed system with many TM, DM and servers.

2.1 Concurrency in virtual environments

. *Virtual enviromet is clearly structured by figure1*.

By the time that the LLT(long lived transaction) reaches the terminal of TM (Transaction Manager), its structure is

revealed to the TM, the requested data within a list is pitched to the transaction scheduler, Transaction scheduler checks for the availability of the data, while negotiating with Database manager if the data is available, the DM (Database manager) creates a temporary cache, this temporary cache contains all the requested data, at this time scheduler acknowledges TM the acceptance of the transaction, thus the transaction can accesses the required data within the temporary cache.

By the time that the transaction is terminated (commit or abort) the temporary cache is also terminated. And the updated data is flushed to the stable storage.

This way if a transaction comes in the wake of LLT, it will access the temporary cache created for the LLT.

Transaction Scheduler also has a pending queue which contains all the conflicting transactions.

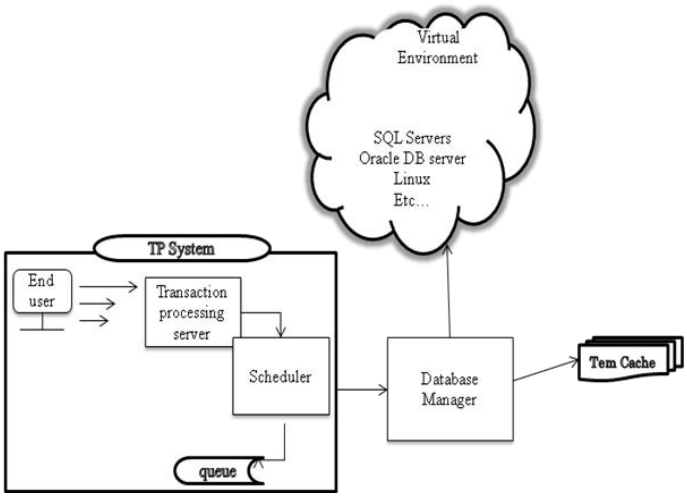


Figure 1, the transaction processing system in virtual environment

Advantages of VE

1. Since database servers share their databases, the maintenance of updates in the system is easier, therefore, consistency of database state is guaranteed.
2. Having centralized TM, DM, and scheduler is less complicated and time saving during transaction processing.
3. System recovery is easier with virtually shared environment since transaction dosn't need to roll-back/roll-forward in so many servers.

2.2 Enhancement of Altruistic Locking (EAL)

We want to further enhance the properties of altruistic locking:

2.2.1 Properties of Basic Altruistic locking (BAL)

1. Two transactions can't hold simultaneous locks on same database object, unless one of them has locked and released the object before the other locked it (second lock holder is in the wake of the releasing transaction).
2. If a transaction (SLT) is in the wake of other transaction(LLT) it should be completely in the wake of that transaction. It means that SLT can't access any data object outside of the wake for that transaction.

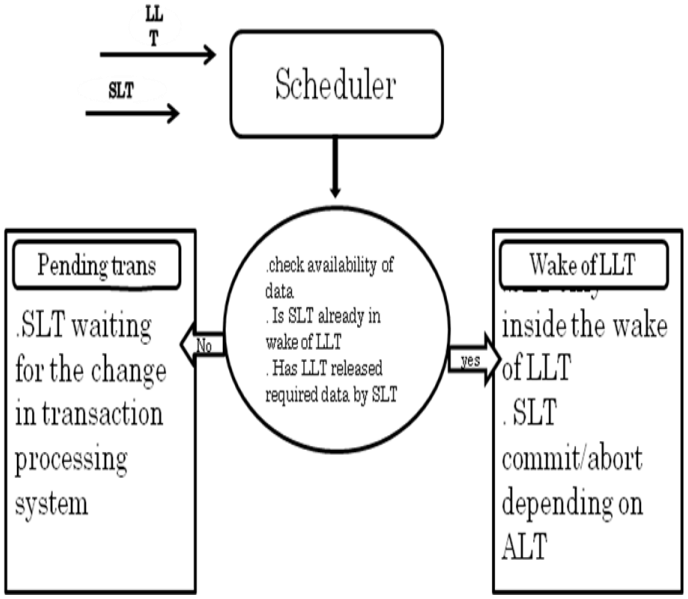


Figure 2, SLT can access data inside the wake of by LLT

2.2.2 EAL (Enhanced Altruistic Locking) properties

In order to represent our terms let's take a look at the following example.

Consider two virtually shared databases :
z[A,B,C,D,E] and ***y[F,G,H,I],***
LLT with two SLT.
Long Lived Transaction list, ***LLT locks [Read(A),Write(B),Reading(C),Read(E)]***
(The underlined C means LLT is now operating on C, and has successfully finished A and B, but has not yet reached E)
.
First Short Lived Transaction list, ***SLT1 try to lock [Read(E),Write(F),Read(A)]***

Second Short Lived Transaction list, ***SLT2 try to lock[Write(A),Write(B)].***

See also figure (3) bellow.

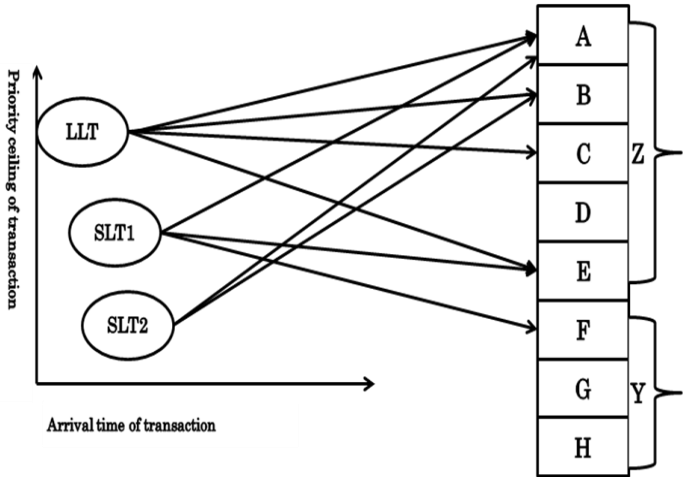


Figure3. Access of transactions, priority is based on arrival time

If we look at SLT1'srequest list, its approach is inside and outside process interval of LLT, according to altruistic locking rule SLT's lock requestcan't be granted.

However SLT2 can be granted, sinceSLT2'sapproach is only insidet hewake of LLT.

According to our conducted experiment to invoke enhanced altruistic locking (EAL), lets consider that the system grants lock request by SLT1.

The result is,

Because SLT1 is a commute transaction with LLT

$$SLT(reqData) \cap LLT(reqData) = Commute$$

It means the action committed by one of these transaction will not interrupt the other.

This proves that: *commute Short-Lived-Ttransaction scan enter inside the wake of long-lived transactions, but they don'tneed to be restricted during LLT' sprocess interval (wake).*

Based on the above assumption we canenhance the Basic Altruistic Locking (BAL) as following:

Enhanced Altruistic Locking's (EAL) properties for long lived transactions in virtual environments

1.Acquire a floppy lock before every read of database object bytransaction.

2.Acquire a solid lock before every write of database object by transaction.

The floppy-lock is a kind of lock by which, the second transaction can enter the wake of the first transaction without following the BAL rule (*restricted during LLT's proces sinterval*), for example, commute transactions.

The solid-lock is the one by which the second transaction has tobe completely under wake of the first transaction and its termination state(commit/abort) depends onfirsttransaction, for example, conflicttransactions.

EAL properties for short transactions in virtual environment

1. Acquire access to the release of floppy-locked data objects without entering the wake of LLT.
2. Acquire access of SLT to the release of solid-locked-data objects only by entering the wake of LLT.

2.3 Protocol

The priority between transactions can be implemented further as bellow:

Pursuing our method EAL combined with Priority ceiling between simultaneous transactions in virtual Environments, The overall protocol is listed bellow in a set of 3 steps:

Step1. The structure of all transactions becomes known to the TM, by the time the transaction arrives at the TM terminal.

Step 2. TM assigns priority based on the arrival time of T.

Step 3. Scheduler initiates and periodically updates the array-Table of current simultaneous transactions prior to their access to manage the priority of transactions. This array-Table contains notation as T[PC(i,j)=k], where T is the specific transaction, (i) shows priority of transaction, (j) is the innings of holding the lock on a single database object and (k) is the number of times T wants to access one specific object (as in table a LLT accesses data object (b) two times so k(b)=2.

Table A, shows the transaction priority and innings.

Transactions	A	B	C	E	F
LLT[PC(i,j)=k]	PC(1.1)=0	PC(1.2)=0	PC(1.3)=1 PCT(1.5)=0	PC(1.4)=0	
SLT1[PC(i,j)=k]	PC(2.1)=0			PC(2.2)=0	PC(2.3)=0
SLT2[PC(i,j)=k]	PC(3.1)=0	PC(3.2)=0			

Furthermore, the acceptance of transaction's attempt of accessing objectis based on the following priorityceiling terms:

1.No higher priority (T) exists for the same object.

2.Access times (k) of transaction over data is zero PCT (i,j)=0.

Commit in this case belongs to the easy calculation of T1(i,j) and T2 (i.j).

For example,PCT1(1,1)>PCT2(2,2) therefore T1which is LLTgets to commit first.

However, in case of PCT2 (2.2)=1 and PCT3 (3,1)=1 don't care since their timing and Required list is different.

The chart bellow shows theoverall structure of our methodology.

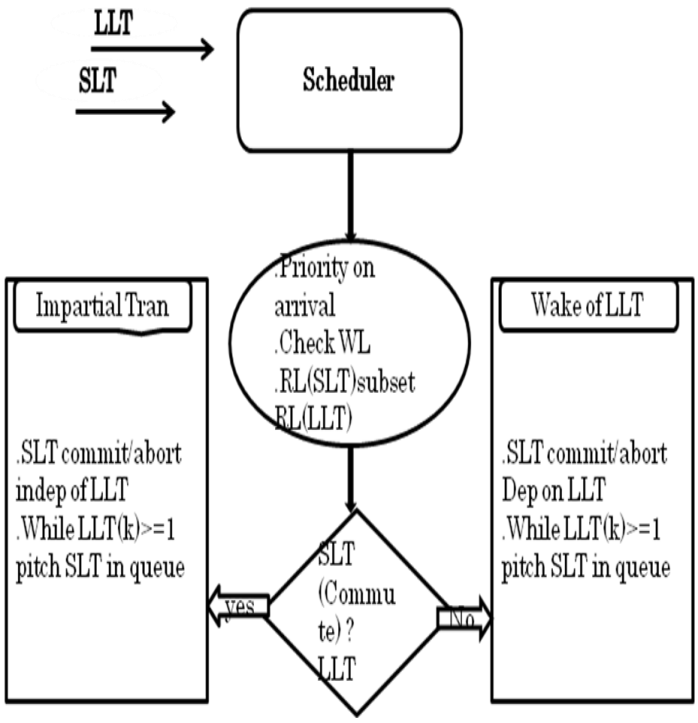


Figure 4, Based on transactions (commute and conflict) situation the structure of BAL is enhanced.

3 Recovery

Recovery rule is obtained by rollback procedure. Our proposed system contains periodical checkpoints, each checkpoint is in charge to backup the state of data (under manipulation by transaction) and over all states of the system from the last the checkpoint. The logging is essential part of this task. If any type of failure occurs in the system, recovery manager rolls back the system to the previous checkpoint in order to omit the aftermaths of failure and to recover the consistent state of the system.

Each checkpoint must maintain three important lists:

(1) active list which contains live operating transactions and live manipulated data object.

(2) commit list, list of the transactions that were committed by the time they reached the specific checkpoint.

(3) Abort list, is a list of transactions which were aborted by the time they reached particular checkpoint.

1. Commit

When the transaction processing is completed at the final server, contents of the cache slot are flushed to the corresponding database at each server.

2. Abort

The abort is caused by the following two cases.

Case1, the fault of the transaction itself occurs at the executing server, despite on the way or final servers.

Case2, the committed transaction read the object x which transaction T1 wrote, at this time transaction T1 is executed normally. However, later, transaction T1 becomes a situation where this transaction T1 must be aborted.

3. Restart

The restart must be carried out for two cases.

Case1, the restart must be executed by the normal periodical checkpoint.

Case2, the restart is also executed in emergency for fault occurrence.

Table 2, concurrencycontrol and recoveryin two different situation (Global Commit and Local Commit).

	Global Commit	Local Commit
Concurrency Control	Keep updated data at cache before commit.	Flush updated data to DB before commit.
	Transaction exists in active list.	Transaction exists in local commit list and local active list.
	Updated log is generated.	Updated log is generated.
	When commit is issued at the final server, commit log is broadcast.	When commit is issued at the final server, commit log is broadcast.
	Flush is done at every server simultaneously.	Flush is done at each server step by step.
Recovery	When abort is needed, abort log is broadcast to every server.	When abort is needed, abort log is broadcast to every server.
	Transaction is removed from active list and added to abort list.	Transaction is removed from local commit list and added to abort list.
	The updated data at cache is not flushed.	The before image of the updated data is flushed into DB (undo).

3.1 Index

Based on the proposed protocol using priority ceiling for the Long Lived Transaction processing, we simulated the concurrency control and obtained experimental results. The concurrency of transaction processing with respect to varying parameters was evaluated, in this case, number of objects located at different servers.

The concurrency of transaction processing is defined by the following equation:

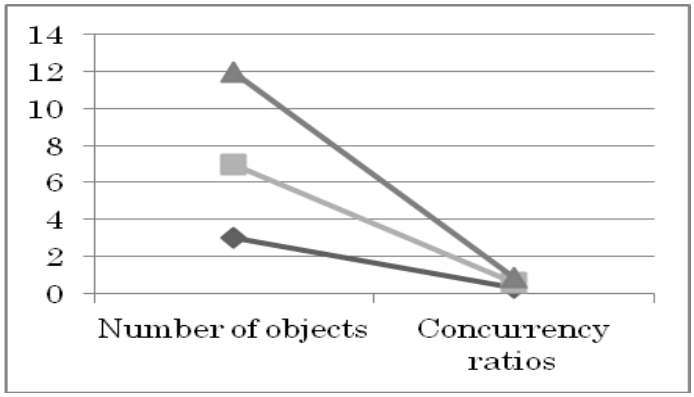
Concurrency of transaction = (number of object) x (issued number of transactions) — (observed access number to objects)

(N x I) – O

Where we assume that, every transaction accesses the same number of objects.

3.2 Experimental Results of Concurrency

The experimental result is illustrated in Figure 5 and 6 also shown in Table 3. As the result of performance evaluation, the concurrency ratio is 27.42 percent for three objects, the concurrency ratio is 28.94 percent for four objects, and the concurrency ratio is 29.02 percent for five objects. The simulation result showed that concurrency meaning the simultaneous accesses for different objects at different server's increases according to the increase of parameters. Meanwhile, for fixed five objects, the concurrency ratios are 28.96 %, 29.02 % and 28.94 %, when the issued numbers of transactions are 1000, 2000 and 3000, respectively. This experimental result indicates the concurrency of transaction processing almost same in spite of increase in the issued number of transactions'.



The figure 5, shows concurrency ratio as per the number of transaction

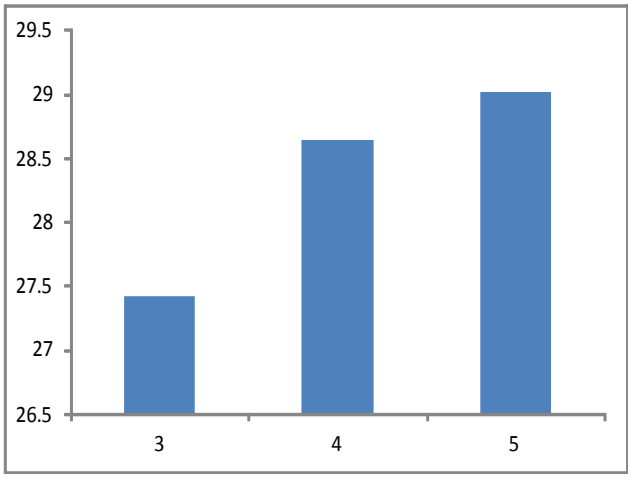


Figure 6, Comparison of concurrency ratios for different number of objects.

Table 3 Comparison of concurrency ratios for different number of objects

Number of objects	3	4	5
Concurrency ratios	0. 2742	0. 2864	0. 2902

4 Conclusions

The enhanced altruistic locking for Long Lived Transaction processing was proposed in virtual environment.
The motive was to promote concurrency in virtual environments with Long Lived Transactions.
The performance evaluation showed tendency of the concurrency ratio. The concurrency of Long Lived Transaction Processing in Virtual Environment was proposed. Priority Ceiling of transactions based on their arrival time was proposed as best option to bring more management in the system.
Aside from that, Recovery mechanisms were highlighted and over viewed.

5 Acknowladgment

Sincere gratitude is hereby extended to those who continued their help until this research was accomplished :
Mr. Kasuke Shuda, Mr. Noriho Okazaki, for conducting and sharing the programing expertise.

Special thanks to the laboratory members of transaction processing, my friends Ryuta Ono, Yuko Nishimura.

For the moral support, thanks to my family and friends who never give up on me.

Above all, utmost appreciation to the almighty God for the divine intervention in this academic endeavor.

6 References

[1] K. Salem, H. Garcia-Molina and R. Alonso, Altruistic locking: A strategy for coping with long lived transactions, Lecture Notes in Computer Science, Volume 359, 175-199, 1989

[2] K. Asai, J. Nishibayashi, K. Yoshihara, M. Nagata, Object-Oriented Serializability in Real-Time Concurrency Control,

[3] M. M. Gore yand R. K. Ghosh Department of Computer Science and Engineering,Indian Institute of Technology, Kanpur, India. Recovery in Distributed Extended Long-lived Transaction Models.

[4] Michael Squadrito and Lisa Cingiser DiPippo and Victor Fay Wolfe. Towards Priority Ceilings in Object-Based Semantic Real-TimeConcurrency Control Bhavani Thuraisingham Department of Computer ScienceMITRE Corporation University of Rhode Island USA

[5] L. Sha, R. Rajkumar, J. Lehoczky, Priority Inheritance Protocols: An Approach to Real-Time Synchronization, IEEE Transactions on Computers, Vol. 39, No. 9, 1990

[6] OZALP BABAO(3LU* *Department of Mathematics, University of Bologna, Piazza Porta S. Donato 5, 1-40127 Bologna, Italy*KEITH MARZULLO** AND FRED B. SCHNEIDER****Department of Computer Science, Cornell UniversiO; Ithaca, New York 14853, A Formalization of Priority Inversion*

SESSION

WEB, INTERNET, AND APPLICATIONS + RELATED ISSUES

Chair(s)

TBA

Integrating HPC Resources, Services, and Cyberinfrastructure to Develop Science Applications Using Web Application Frameworks

M. P. Thomas[1,2], C. Cheng[2], S. More[2], and H. Shah[2]
[1]Department of Computer Science, San Diego State University, San Diego, CA, USA
[2]Computational Sciences Research Center, San Diego State University, San Diego, CA, USA

Abstract *– The Cyberinfrastructure Web Application Framework (CyberWeb) simplifies access to heterogeneous, computational environments required by high-performance computing applications. CyberWeb has three core components: a Pylons Web2.0 framework, including XML, JavaScript, AJAX, Google APIs, social networks, and security; a Database and Web interface for configuring installations, applications, users, remote services; a job distribution service framework for task execution and management. CyberWeb design philosophy includes: "plug-n-play" mode - applications dynamically discover modifications to the system and automatically reload new components; "non-invasive" philosophy - no software is required to be installed on remote resources, it interfaces existing software and services. CyberWeb supports basic job functions (accounts, authentication, execution, task history); operates in heterogeneous environments from remote large-scale systems (XSEDE/TeraGrid) to local systems; applications built on top of the core framework (ocean, thermochemistry, education). In this paper we present the CyberWeb architecture, highlighting the database and JODIS architectures, and demonstrate its usefulness with application examples.*

1. Introduction

Advances in technologies and languages used for parallel and distributed computing have often presented the science researcher with the challenge of migrating a model or application to ever increasingly complex systems. Often, many science applications require tremendously large compute, data and archival resources and high-speed networks in order to manage results, or the model is often legacy code that is stable, but no longer ports to the high-end systems that are available. Cyberinfrastructure (CI) integrates hardware and software for computing, data management and information retrieval, visualization and analysis using interoperable software/middleware and services that are based on Web and Internet technologies. The NSF's *Cyberinfrastructure Framework for 21st Century Science and Engineering (CF21)* sets an ambitious goal that next generation of cyberinfrastructure software must seamlessly couple high-end and low-end CI resources, networks and services, with users and applications using these commodity Internet and Web technologies [1]. The US DOE held a workshop in 2008 addressing the grand challenges and limitations that exist today in the field of high-resolution climate and Earth modeling systems [2]. The outcome of these efforts have helped to define the requirements for the next generation of HPC applications: new/updated models are needed that can take advantage of these cyberinfrastructure based environments; the NSF and DOE need to construct and support this cyberinfrastructure over the long run; and new tools and libraries are needed to facilitate the development of these applications.

There are many efforts to develop common tools that can be reused at all layers of the science gateway architecture, including services to resources (data, compute, visualization, network), middleware, and user interfaces and portals and science gateways. Science gateways (and computational environments) are terms used to characterize the systems and tools that facilitate the utilization of CI by science applications; gateways typically involve a user interface [3][4]. This research is part of this effort: the software developed is contributed to the suite of tools developed by the NSF funded Open Grid Computing Environments (OGCE) project [5], which focuses on the development of gateways and tools. In this paper we describe advances made the SDSU Cyberinfrastructure Web Application Framework (CyberWeb, [6]) which is designed to simplify the development of advanced computational environments (CEs) used by high-performance computing (HPC) applications and science gateways. CyberWeb improves on standard CI toolkit functions (job execution, account management, task history, GSI authentication, etc) by hosting all applications as Web services, portal Web pages, or Web 2.0 gadgets. CyberWeb is being used to develop applications that need to operate across large-scale grids such as the XSEDE/TeraGrid, local university clusters, and commercial or public systems.

In this paper, and overview of the CyberWeb architecture (Section 2), highlighting the database and JODIS architectures are presented. In Section 2.5, installation and deployment experiences are present, and Section 4 presents application examples.

2. CyberWeb Architecture

The Cyberinfrastructure Web Application Framework (CyberWeb) architecture is shown in Figure 1. It reflects the standard 3-tier architecture found in systems that connect clients (human, computer) to remote resources via middleware. Front-end clients can be applications, services, or human (browsers, desktop apps, command line interfaces). The backend tier (cyberinfrastructure) includes local services, web services, other applications, and remote

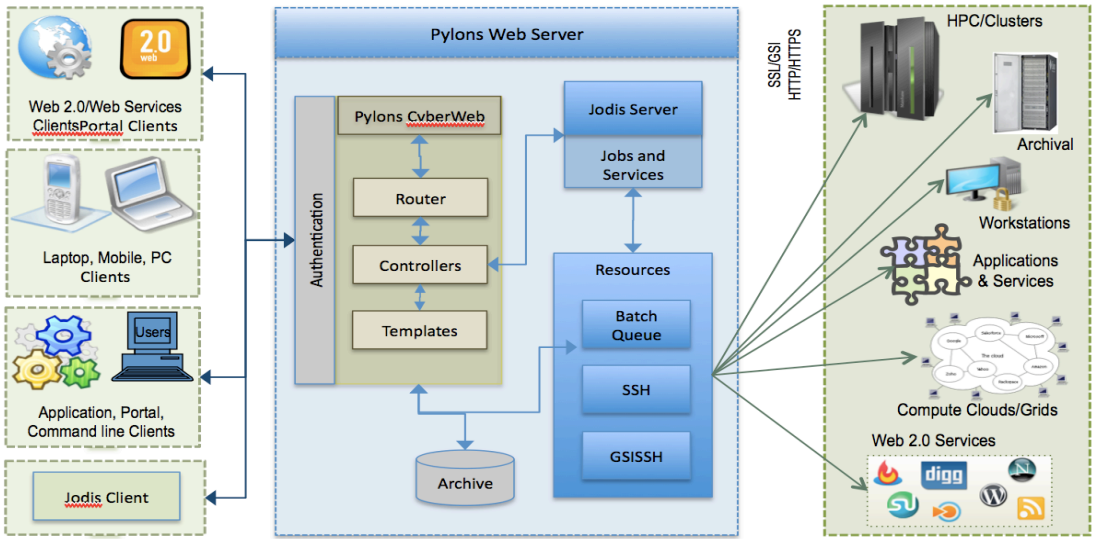


Figure 1. CyberWeb Architecture Showing client, middleware and backend resource cloud. Middleware layer includes the Pylons framework, the CyberWeb database and Jodis services [6].

grid, computing and data services. CyberWeb uses the Pylons Web Application Framework [7] for its core Web 2.0 services.

2.1 Pylons Web Application Framework

Pylons is a component based, lightweight Web 2.0 application framework that using only Web 2.0 technologies, including WSGI (Web Server Gateway Interface, relational databases, XML, JavaScript, AJAX, Google Gadgets, social networks, and security. Any number of components and libraries developed by other projects are part of the system and new ones are easily integrated. Pylons uses the model-view-controller (MVC) request-response architecture (Figure 1). The *Model* contains the data used by applications. Often the model refers to database tables. The *View* reads the data from the model and displays it to the user. The *Controller* manages the logic of the application, activates views to display data to the user, or parses information from the user and stores them to the models. CyberWeb application developers work with all three components. The MVC approach decouples the services layer from the logic of the code behind it, allowing the same application to be hosted as different service types (Web service, Web page, XML request, iGoogle gadget). Pylons supports dynamic routing via Routes (Python version of Rails), and is used for mapping URL's to Controllers/Actions and generating URL's. Routes makes it easy to create "pretty" and concise RESTful (REpresentational State Transfer) URIs (uniform resource indicators). Template packages are used to build dynamic Web pages from a variety of sources: HTML file, XML generators, scripts (perl, php), or templates.

2.1.1 Key Features

Pylons has several features that are beneficial to science application and gateway developers. A key philosophy of the Pylons project is external components can be plugged into the framework with only the minimal code-base necessary. This facilitates customization. Although the features of Pylons are as rich as the libraries and modules that are available to run with it, a few key features of importance to this research are listed here.

Routing and "Beautiful URLs:" The feature makes it easy to create "pretty" and concise RESTful (REpresentational State Transfer) URIs (uniform resource indicators). With RESTful URI's, the information being transferred is stateless, and independent of resource details.

SQLAlchemy database: Pylons supports many databases, including SQLAlchemy which is a Python SQL toolkit and Object Relational Mapper (ORM). ORMs map the database structure to objects.

Interfaces to multiple template packages: Templates are used to build Web pages in a dynamic manner. Pylons allows any type of HTML file, XML generators, or template modules to be used. xx This allows application developers flexibility in choosing how they want to build their application.

Dynamic update and interactive debugger: An important feature of Pylons is the "reload" feature: it supports dynamic class loading for all MVC layers; modifications to the code cause the compiler to update the binary and re-load it into the server while keeping the server live.

Web 2.0 and Open Social Interfaces: Pylons' RESTful Web services allows Pylons components be published as services, widgets and gadgets, and desktop applications. Pylons interfaces to other toolkits including: Google Application Engine and OpenSocial gadgets; PyWebKitGtk (an API for developers to program WebKit/Gtk).

In addition, AJAX/JavaScript support for libraries such as the Yahoo! UI Library (YUI). The YUI, CSS, AJAX, and Javascript are used by CyberWeb for nearly all demo web pages.

2.2 Model and Database Services

At the core of CyberWeb is its database system, where most of its configuration data is stored (users, hosts,

Table 1. List of CyberWeb model category and table names.

Table Category	Table Names	Description
Group	Users, Groups	CyberWeb user accounts (chosen by user), Groups (used for authentication & authorization; e.g. admin, developer, application)
Accounts	Account	Accounts on remote resources; owned by a CyberWeb user account.
Protocol	Protocol	HTTP/HTTPS, TCP/IP, SSH, GSISSH,
Services	Service Type, Name, Service	Used to define service types (authentication, queue, application, archival); names of specific services (SSH is type authentication); a Service (e.g. an SSH Service) has a type, name, and is installed onto a Resource.
Queue Systems	QueueType, Name, Info QueueService	Used to define queuing service types (batch, condor, grid); names of specific services (LSF, PBS, Torque, SGE); a QueueService (e.g. an SGE Service) has a type, name, infor, and is installed onto a Resource.
Resources	Resource	Defines compute, archival, and networks used (primarily) for remote job execution. A Resource has DN or IP address; and are used (typically) to host Services.
Job	Job	CyberWeb tracks internal Jobs and Tasks, maintaining job history. This is independent from job ids used on remote resources.
Message	MessageTypes, Message	Used for communication among CyberWeb users and applications, messages types include news, events, jobnotification; a Message has a type.

authentication, jobs, job history, etc.). In addition to using this database to store session and user information, a key design goal was to develop a database architecture must be easily modified, populated dynamically, and then be used to configure and build virtual organizations (VO's). Furthermore, the system must operate in a "plug-n-play" mode: once modifications are committed, the new information be readily available to the relevant components of the system. This includes adding compute and archival resources, defining and naming services, and configuring these services on a resource. Once set up, all internal services use this database to discover active resources, the operational status of services, and the ACL of a CWuser for that service or resource, and relevant user preferences.

2.2.1 Schema Design

The schema design was based on an evaluation of schema used by multiple projects (TeraGrid [8]; Open Grid Forum [9]; and W3C/IETF standards). The requirement that the schema be simple, and useful to non-database experts developing small applications, drove the design approach to simplify and reduce the number of tables and elements in them to a minimal set. The assumption is that an application developer can expand the initial database to meet project requirements. The perspective is from the point of view of a high-performance computing environment: a resource is typically a host computer (cluster, archival); services run on these resources (SSH, FTP); and users have access to them via authentication such as username/password or GSI certificates.

The database is initialized using a JSON input file, and can be populated dynamically. The database design has four key components: (1) SQLAlchemy: a Python SQL toolkit and Object Relational Mapper (ORM) that interfaces to multiple RDBs; (2) SQLite: a lightweight, easy to install database, the data is stored in memory within the server (note that CyberWeb has also used MySQL, which is desirable for larger databases); (3) The Model is coded into a python module, and loaded into the Pylons server at startup; and (4) Data is initialized using a flat text file written in JavaSript Object Notation (JSON) format, and easily edited to seed the database.

Because the model is defined in an ASCII file, the developer can easily add new or modify existing tables to the initialization database or make modification at run-time. Note that for the examples shown in Table 1, all examples can be modified or new ones added via the Database Admin Interface (CW-DAI), described below. However, caution should be used to ensure that core services are not adversely affected. A good practice is too define all core components and services in the initialization database.

If the codebase is loaded into a developer environment, the file can be viewed easily with JSON markup tags; if using an SVN, then an SVN viewer (eg. TRACS) allows for easy viewing of the model. This is a key advantage for developers who are not database experts. As part of the codebase, there is a demo portal which contains examples of how to query the model (database) for common tasks.

2.2.2 Database Admin Interface

The database can be managed via the SQLite (or MySQL) command line interface. CyberWeb also has a comprehensive Web based Database Admin Interface (DAI) that is used to dynamically add, delete, modify, configure and test resources, the services that will run on them, and a users/groups access control list (ACL). The DAI module is designed using the very Rich User Interface design. User Interface guides user to do operations like add, edit or delete on any entit, along with JQuery, Ajax, and javascript. Figure 2 captures a usecase of the workflow used to configure and activate accounts for CyberWeb users (CWusers) on a remote compute resource.

CyberWeb securely manages accounts (using HTTPS, and sensitive data is encrypted and stored outside Web space, and no passwords are stored), maps CWuser accounts to accounts on remote resources, and tracks preferred authentication schemes (e.g. PKI/SSH, GSI/SSH). All users have PKI serverside credentials; GSI credentials can be uploaded or obtained from a MyProxy server. For PKI authentication, CyberWeb configures a password-less SSH service for the Cyberweb user. To activate the resource for the CWuser, a secure test is performed using an SSH command. The CWuser must have an account on the remote resource.

2.3 Job Distribution Service (JODIS)

The job distribution Web service framework (JODIS) was allows CyberWeb applications to distribute jobs across several campus compute clusters each running a different resource manager and each controlled by a different system administrator. Its main duty is to distribute application workloads across heterogeneous computing systems by abstracting middleware and resource management systems. The JODIS Web Service application framework based on the master/worker design pattern using common commodity software allows us to bridge these systems unbeknownst to the developer and user. It has been tightly coupled with the SDSU Cyberinfrastructure Web Application Toolkit (cyberWeb) framework making it a full-scale web application [10].

2.3.1 JODIS APIs

Job Management: The job management API is the one component of JODIS that does not ease development by abstracting a layer in the application stack, instead, this API tracks and manages jobs run with this system. It aggregates every aspect of the job from what kind of job, how many tasks, when it was run and the run time. We believe all of these aspects are useful for the administrator of the portal and will ultimately be used to measure how well an application is running.

Job Queue: This API allows the developer to interface to any queuing system. The current system includes Condor, PBS and Sun Grid Engine (SGE). There are many more in use at SDSU and elsewhere. Each application has varying syntax and command-line parameters. The JODIS job queuing API abstracts these allowing the developer to make one call to a system's queuing application regardless of what that may be. This allows a developer to quickly move to a new system or handle system configuration changes.

File Transfer: The second API that JODIS provides is file transfer. The API mimics that of a secure copy command. This API allows a developer to move and copy files from one machine to the next regardless of the protocol you are using to connect. A big use case for this functionality is easy on-the-fly deployment of an application and transferring data as needed by your application. A second common use for this is allowing a user to access his/her data results and move it back to another machine possibly for visualization, archiving or sharing with colleagues. This file transfer view method discussed earlier in the paper extensively utilizes the JODIS file transfer API.

Raw System Calls: Jodis allows the developer to run raw commands on the remote resources as if he/she were at the command prompt. This gives the developer the ultimate flexibility when developing applications using CyberWeb and JODIS. A user can call the raw method and pass in the command to run. Unfortunately, this is done synchronously. The user must wait for the command to return before an HTTP response is sent back to the user.

2.3.2 JODIS Services

JODIS consists of multiple services offered through a Web 2.0 server environment. Providing each component as a service allows JODIS to scale horizontally. In the web application environment, these services work together to provide end-to-end job dispatching service simultaneously to multiple clients. The architecture for the JODIS system can be found in Fig. 1. The Web server environment (based on CyberWeb, see below) provides users with methods of communicating with JODIS using either a Python client API or Web Service to access the Job Service.

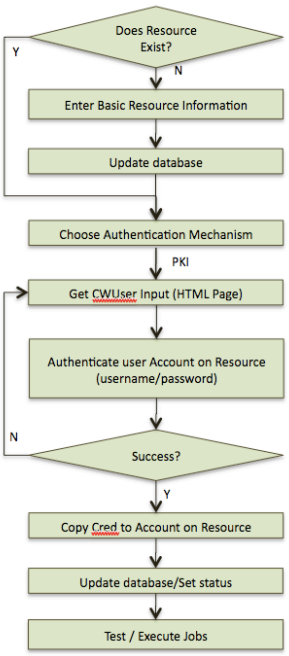


Figure 2. Admin flowchart for configuring user accounts on a resource.

The Web Service allows a wide variety of applications to interact with JODIS regardless of location, device or programming language. The Job Service provides a provides a majority of theuser-accessible function calls and manages a user's jobs regardless of the resource. The Resource Service works on the back end as a singleton to manage the various connections between the JODIS and the compute resources being used. The ability for JODIS to gather usage information and use this data for predicting job runtimes and for selecting where to run a job provides a useful approach to running MTC jobs.

Job Services: Clients primarily interact with the JODIS Job Service API. It is responsible for integrating all the services that make up Jodis. This service is used for job submission and monitoring. This service wraps the the resource management systems and abstracts the complexities involved with job submission such as job syntax, tracking and management of all jobs across the different resources. CyberWeb interfaces to batch queue systems (LSF, PBS, SGE) and schedulers (Condor, SGE). JODIS uses a job runtime "Guesstimation" to forecast job runtimes and a distribution policy to dynamically choose which compute resource to use for each job.

Resource Service: The resource service provides one essential functions to JODIS and that is controlling middleware. The resource service offers communication to the compute resources and client targets via Secure Shell (SSH) or GSI-Enabled SSH. This service leverages the CyberWeb database for fast and flexible storage of resource metadata, user account information and access control data. This service can also be used to stage files on the various compute resources and target machines.

Client Services: JODIS hosts a general client Web Service for authorized job submission. Clients interact with JODIS directly using the Python API, or more popularly,

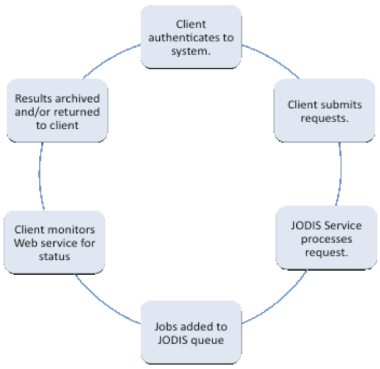


Figure 3. Diagram of a typical JODIS job cycle for job submission from a CHEQS client.

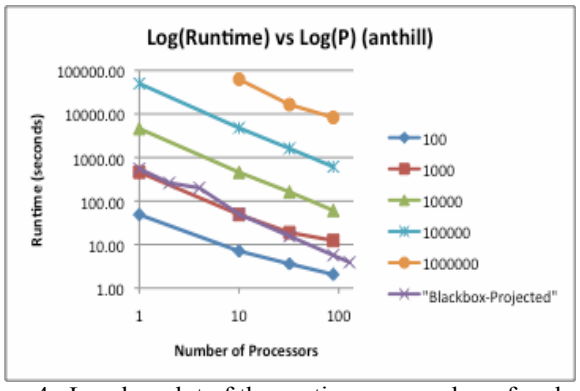


Figure 4. Log-log plot of the runtime vs. number of nodes as a function of the number of tasks on anthill.sdsu.edu [15].

through the RESTful Web Service interface. JODIS also hosts a WSDL that allows users to find the service as well as keep up-to-date on the latest API.. Developers can extend zzthe client service for specific applications with the use of the Job Builder Client interface. The client service can hook into the JODIS job service to provide functions such as pre-processing, post-processing and more. The client service is responsible for clustering the tasks, for each resource, which are then passed as a collection of jobs to the JODIS job service and distributed. An example of the Jodis job cycle can be found in Figure 3. An example of the client service is described below in Section 3.4.

2.4 Authentication

CyberWeb security works at two levels: requests and responses processed by the Pylons Web server; and transactions performed on behalf of a CWuser on a remote resource or service using PKI or GSI authentication. To authenticat users, CyerWeb uses the Pylons AuthKit module [7] which is a complete authentication and authorization framework for WSGI applications, and was written specifically to provide Pylons with a flexible framework for managing these tasks. This module queries the user table in the CyberWeb database. Once the user is authenticated, the user's ID and group information is stored with the user session data. Based on the ACL for an application, CyberWeb automatically requires the user to re-authenticate their session after a 5-minute period of idle time.

Decorators have been built using AuthKit to allow developers to permission applications based on the user or the group of a user. These decorators wrap access to the decorated method and redirect the user to the login page or a 404 – "permission denied" page depending on whether the user is not authenticated or lacks permission. The advantage to using these decorators is that the developer has fine-grain control over access and can control user interaction based on the application. For eample, a user might be able to create a workload to be run, but must have permission before the user can submit this workload.

For remote transactions, two methods are currently supported: Public Key Infrastructure (PKI) and Grid Security Infrastructure (GSI) developed by the Globus project [11]. PKI is used for *ssh* (secure shell) transactions. Users must set up passwordless access in order to use a resource (see Section 2.2.2). A credential associated with the server is created for all CWuser accounts. If the user has an account on a resource, then CyberWeb will put a copy of the credential into the appropriate file on the remote machine. For grid based transactions, CyberWeb uses the MyProxy API [11] to obtain a user proxy certificate, which is stored out of Web space, and *gsissh* to connect to remote resources for job execution.

2.5 Installation and Deployment

CyberWeb applications can be run from most modern-day operating systems: it has been tested on linux, Windows and Mac OS X. CyberWeb is written using the Pylons web framework. It requires minimally two Python libraries a) Pylons, the web framework core of CyberWeb, b) Paramiko, a Python wrapper for the SSH library. After installing the pre-requisites, download the CyberWeb source code: it can be run from any directory on your machine. An installation challenge is where the CyberWeb server is located. Network access to resources behind firewalls is a common issues. For example, in order to access the clusters at San Diego State University, servers must be on the campus network.

A key aspect of the OGCE project is its use of Maven [12] to install the entire framework and all software dependencies automatically for the client. In Python, this is done using the Easy Install package. Easy Install automatically downloads, builds, installs, and manages Python packages. It installs the tar or jar equivalent called the Python Egg. The Easy Install software comes with commands (configured by Pylons) that allow you to bundle your application into an egg for distribution. Earlier experiments with extending the Paste installation egg to include all software and versions needed for a complete CyberWeb package were successful and this will be done for future releases. The system administrator will be able to fire up CyberWeb out of the box. Settings for the CyberWeb installation can be found in the development.ini or production.ini depending on the environment. In many aspects, these two files will mirror each other. The main difference between the two should mainly be the debug variable. The production.ini file should have debug set to

false. This prevents the stack trace from being displayed when an uncaught error is encountered.

Straight out of the box, the development.ini file directs CyberWeb to create a folder in the top level directory of CyberWeb to store user data files. The top level directory is referred to by the variable *"%(here)s*". This CyberWeb data directory stores users' public/private keys at the top level and user directories and data in a directory below. The top level CyberWeb data directory is not accessible via CyberWeb or Paster, which is the same level of security that the linux operating system uses to store these keys.

New Pylons projects are built using Paste - a Web development and application installation tool similar to Ant or Make. It creates python middleware modules (router, config, and mapper template files), a simple python Webserver that can process WSGI requests, and the directory structure needed for a full site (to be populated by the developer). A single command installs the application template (all the codebase needed including the directory structure, basic files, and data to support a new portal): *%paster create –-template=pylons* pyGateSite. The portal server is started using the following command: *$paster serve –reload development.ini*, which lauches the welcome page. Alternatively, you can install all software into a "virtual environment" which is essentially an isolated working directory with all libraries contained within it. This allows developers to work independently of the host operating system, to keep stable versioning control, and locally manage development tasks. This facilitates distribution and deployment of a project. This will be useful for providing a reusable toolkit or deploying a Pylons application to a cloud computing resource.

3. Application Examples

CyberWeb has been used to develop Web services, Social gadgets, and portals in the biology, geospatial mapping, and ocean application areas. The examples below demonstrate client-server use of Web services, portals, and visualization Web pages.

3.1 CyberWeb Demo Portal

As part of our development of CyberWeb, a user portal has been developed that demonstrates how to build CI enabled applications and for testing services. Figure 5 is a composite image of the UCOAM portal (see Section 3.2), and many of the components are contained in the demo portal The portal system is intended to be used as a startup application for new projects. The portals capabilities include: secure login with group access control to selected pages; user and account management with the MyCyberWeb customization section; MyProxy credential management; job tracking and history; and data management using the Jodis file transfer widget; interactive Web pages for a Job execution and file management that are also part of a planned unit test system for checking resources and services. The portal also hosts the Database Admin pages. Through the myCyberWeb interface, a user can update preferences, send/receive messages, track jobs, and configure authentication on remote resources.

3.2 Unified Curvilinear Ocean and Atmospheric Model (UCOAM)

The UCOAM model differs significantly from the traditional approach, where the use of Cartesian coordinates forces the model to simulate terrain as a series of steps. UCOAM utilizes a full three-dimensional curvilinear transformation, which has been shown to have greater accuracy than similar models and to achieve results more efficiently [13] [14]. CyberWeb is being used to develop a computational environment for the UCOAM project with the following features: community access portal for expert and non-expert users (see Figure 5); running and managing jobs, and interacting with long running jobs; managing input and output files; quick visualization of results; publishing of computational Web services to be used by other systems such as larger climate models [**Error! Bookmark not defined.**] [10].

3.3 Data Viewer Tool

The Data Visualization tool is a custom application that is being developed for the UCOAM application. provides portal users with the ability to run a suite of post processing tasks including analysis of parallel performance, and model simulations. The Data Viewer utilizes several CyberWeb components (database, JODIS), and emerging technologies including python, pylons, AJAX, javascript, jQuery, gnuplot and gnuploy.py library. The tool allows users to select a job through the data browser and then to run the Job Analyzer. The Job Analyzer presents a job summary and a variety of plot options. These plotting options support different plots for viewing timing/performance, or contours of the velocity, temperature, or and pressure changes (see Figure 5). The plotting options are stored dynamically, and are extensible. Once the user selects any particular plot, an Ajax call is sent to the server, which uses Jodis to send commands to the archival host to create the image on remote using gnuplot scripts. This image file is then returned to the client browser in the form of data-bytes for display.

3.4 CyberCHEQS

This example highlights the use of JODIS and CyberWeb, we developed a simple job distribution Web service (Jodis), which runs a many-task computing (MTC) [15] jobs for the CyberCHEQS thermochemistry applications [16]. The tasks were run in a heterogeneous environment (Figure 4), using TeraGrid and SDSU machines simultaneously, on hundreds of nodes, moving data and results between remote resources. Using CyberWeb, the resolution of Flame3D was increased from 10^3 to more than 10^6 control volumes and significant reduction in run times (by a factor of over 40 for a large test case of 128 processors and 10^7 tasks)

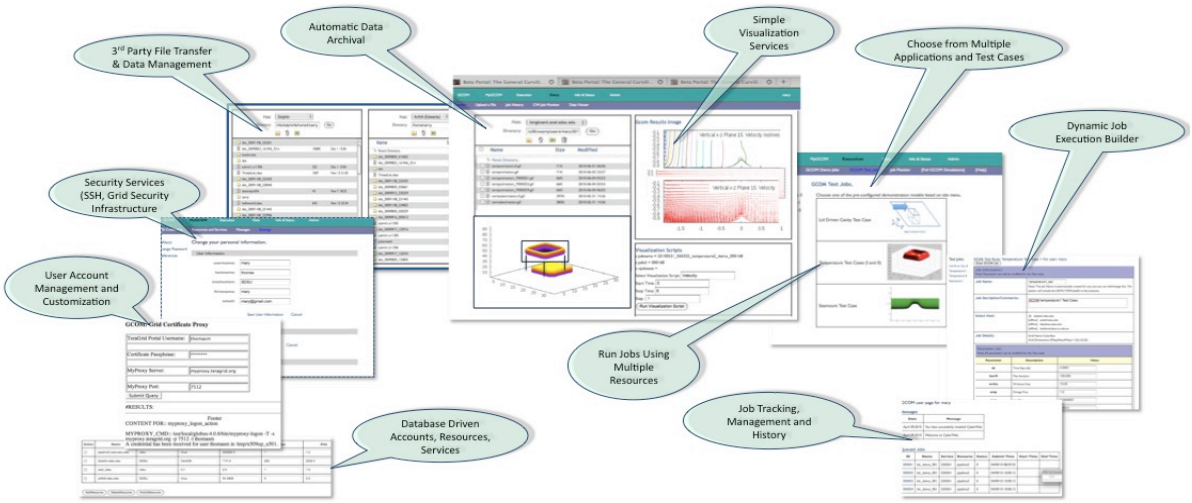


Figure 5. The UCOAM Simulation Portal. Composite image shows file management, user/account management, simulation job submission and history, and visualization [**Error! Bookmark not defined.**].

4. Conclusions and Future Work

CyberWeb applications have been developed that demonstrate its usefulness for client-server access to Web services, portals, and visualization Web pages. CyberWeb is able to operate in a "plug-n-play" mode: applications dynamically discover modifications to the database, check the service status, and when available, use that service or resource. It has a "non-invasive" philosophy: no software is required to be installed on remote resources; rather interfaces using existing software and services are created. CyberWeb has been used to develop applications in ocean modeling, thermochemistry, and education; it has been used to access large-scale (EXSEDE/TeraGrid) and local compute and archival systems.

Future plans include enhancing or developing: visualization tools; interactive job management; full third party file transfer; integration of cloud computing resources; interface to modifying the initialization JSON data for the database; addition of new authentication systems such as OAuth or Kerberos (both are used among in the scientific Cyberinfrastructure community); additional queuing systems. Plans are underway for the software to be bundled into a Python egg, and the software will be added to the NSF funded Open Grid Computing Environments (OGCE) project [5], which develops gateways and tools.

5. Acknowledgements

This work was supported in part by the National Science Foundation (Grants #0753283, #0721656), the Department of Energy (DOE # DE-GC02-02ER25516), and with resources available with an NSF funded XSEDE (TeraGrid) allocation (TG-CCR110014) and the San Diego State University Computational Sciences Research Center.

6. References

[1] NSF Vision: Cyberinfrastructure Framework for 21st Century Science and Engineering (CF21). Available at: http://www.nsf.gov/pubs/2010/nsf10015/nsf10015.pdf

[2] Washington W. Challenges in Climate Change Science and the Role of Computing at the Extreme Scale. Proc. of the Workshop on Climate Science, Nov., 2008, Washington D.C.

[3] Wilkins-Diehr, N. 2007. Special Issue: Science Gateways—Common Community Interfaces to Grid Resources: Editorials. Concurr. Comput. : Pract. Exper. 19, 6 (Apr. 2007), 743-749.

[4] Alameda, J, M., et. al., The Open Grid Computing Environments collaboration: portlets and services for science gateways. Concurrency and Computation: Practice & Experience, Volume 19 Issue 6, pg.1078, 2007.

[5] NSF NMI Open Grid Computing Environments (OGCE) project. Website last accessed on 1-Jan-06 at http://www.ogce.org.

[6] Thomas, M. P., Cheng, C. Development of Web Application Frameworks for Cyberinfrastructure. CSRC Technical Report, 2010.

[7] Pylons Web application Framework Website, Available: http://pylonshq.com/

[8] The TeraGrid MDS4 Schemas Project Page: http://dms.teragrid.org/mediawiki/index.php?title=MDS4_Schemas

[9] Open Grid Forum GLUE 2.0 XML Schema project page: https://forge.ogf.org/sf/projects/glue-wg.

[10] Thomas, M. P., Castillo, J. E., Development of a Computational Environment for the General Curvilinear Ocean Model, 2009 J. Phys.: Conf. Ser. 180.

[11] The Globus Project. Available: http://www.globus.org .

[12] Maven Software Project. Available: http://maven.apache.org/

[13] Abouali M., Castillo J.E., "Unified Curvilinear Ocean Atmosphere Model (UCOAM): A Vertical Velocity Case Study", Journal of Mathematical and Computer Modeling. (Accepted on March 17th, 2011), DOI:10.1016/j.mcm.2011.03.023.

[14] Mary P. Thomas, M. P., Castillo, J. E. "Parallelization of the 3D Unified Curvilinear Coastal Ocean Model: Initial Results." Accepted for publication, International Conference on Computational Science and Its Applications, 2012. ICCSA '12.

[15] Thomas, M. P., Cheng, C., Edwards, R. A., Paolini, C. P. Improving the Performance of Thermochemical Computations Using Many-Task Computing Methods. CSRC Technical Report, 2010.

[16] Paolini, C. P. and Bhattacharjee, S., A Web Service Infrastructure for Distributed Chemical Equilibrium Computation, Proceedings of the 6th International Conference on Computational Heat and Mass Transfer (ICCHMT), May 18–21, 2009, Guangzhou, China.

An Improved Cache Mechanism for a Cache-based Network Processor

Hayato Yamaki[1], **Hiroaki Nishi**[1]
[1]Graduate school of Science and Technology, Keio University, Yokohama, Japan

Abstract – *Internet traffic has increased due to the development of richer web content and services. In particular, IP telephony, Messengers, and Twitter are composed of a large number of small packets, and these services are expected to increase. Thus, routers need to handle large-bandwidth fine-grain communication. We proposed a network processor called P-Gear, which has a special cache mechanism that reduces the processing load by taking advantage of the localizations of network traffic. In this paper, we describe cache architecture for P-Gear and we propose an enhanced cache mechanism, Multi-Context-Aware Cache. Multi-Context-Aware Cache can improve the cache-hit ratio because it can control cache entry by analyzing the packet's header. We implemented P-Gear as software and we simulated this mechanism using real network traces. The simulation showed that Multi-Context-Aware Cache is effective in reducing the processing cost of processing arrays on a network processor.*

Keywords: Cache-based Network Processor, Layer-7 Analysis, Network Processor, Network Traffic Engineering

1 Introduction

Internet traffic has increased due to the popularization of personal computers, smartphones, and growth of richer web contents and services. Furthermore, services that are composed of a large number of small packets such as video traffic, IP telephony, and Messengers are expected grow in popularity. If these enormous fine-grain packets are concentrated in a router, its processing capacity might exceed its limit. Future backbone network router need to handle large-bandwidth fine-grain communication.

Conventional network processors utilize the characteristic independency of packets, where each packet can be processed independently by a Processing Unit (PU) or in parallel by increasing the number of Processing Units (PUs) or by implementing a multithreading mechanism to achieve high throughput packet processing [1][2]. However, the leakage of current or electrical power consumption makes it difficult to improve the number of PUs accumulated due to the scaling rule of semiconductors. Therefore, it is necessary for network processors to reduce PU processing costs.

In earlier research, a mechanism was proposed that the reduced costs of packet processing when retrieving routing tables [3][4][5]. This mechanism resolved the processing when retrieving routing table by using cache architecture. Thus, we proposed an architecture for a network processor, called P-Gear, which was an extension of this mechanism [6][7]. Eliminating the cache-miss ratio is significant for P-Gear and all cache-based network processors. P-Gear takes advantage of the temporal locality of network traffic to reduce processing costs like general network processors. It caches to process the packet, which improves the potential of cache-based network processor. However, the cache-hit ratio depends on the number of processing units required and the cache memory size. Reducing the cache-miss ratio is an important issue for P-Gear type network processors. In this paper, we propose and evaluate a mechanism that allows us to control cache entry by focusing on trends in network traffic to produce a lower cache-miss ratio using the same cache entry size.

2 Related Work

Various cache-based network processors have been proposed in earlier research. These processors exploit the temporal locality of network traffic, where many sequential packets arrive during a short period of time. In previous studies [3][4], a cache architecture for the high-speed processing of routing table lookup on layer 3 was proposed. This architecture utilized a quarter of the L1 cache to store the destination IP address and the results of routing table lookups. Part of the destination IP address was used as the virtual address of a cache. The ID of an output port was obtained when the cache was hits. Moreover, they considered that the cache entry size, block size, and associativity size were important and they proposed a method for compressing the entry size of the routing table [8]. In paper [5], another cache mechanism was proposed for routing table lookup, which was evaluated using real network traces. This study showed that it was effective to use the upper half of an IP address as a cache tag and the lower half of an IP address as an index in the cache.In another study [9], the cache architecture was partitioned according to a prefix length for routing table lookup. They demonstrated that IP traffic had temporal and spatial locality.

Thus, cache-based network processors are used for resolving the lookup of routing table entry. However, because of the emergence of rich services, recent network processors are required to execute a lot of complex processing in addition to routing table lookup, e.g., packet encapsulation, and these processes may be a bottleneck in packet processing. Thus, we proposed a cache-based network processor called P-Gear, which facilitates high-speed processing for various processes by caching the IP address, port, and protocol.

3 P-Gear

In this section, we provide an outline of P-Gear. P-Gear utilizes two features of packets. The first is that packets are processed by the same process in the same session. The second is that packets often arrive continuously in a short period. P-Gear caches before processing the packet and applies the cached process to later packets during the same session. The later packets in the same session can be processed without using PU. P-Gear defines a five-tuple (Source IP, Destination IP, Source Port, Destination Port, and Protocol) as one session. The architecture of P-Gear is shown in Figure 1.

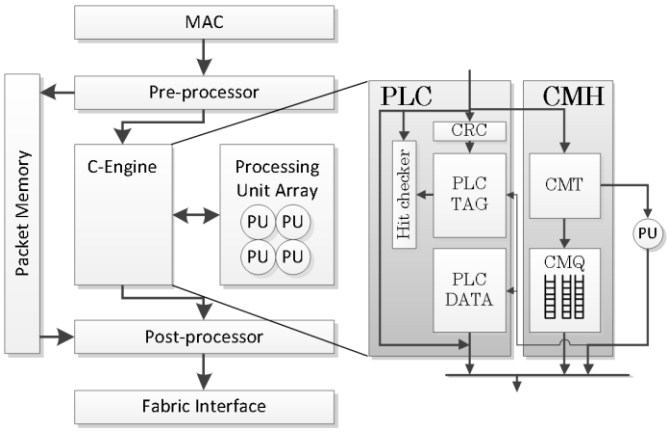


Fig. 1 Architecture of P-Gear

When PLC hits, P-Gear can process the packet without using PU by applying a cached process to packet processing. If there is a cache-miss, the packet is sent to the Cache Miss Handler (CMH), and P-Gear processes the packet normally using PU. After processing the packet, the session information of the packet and the results of the packet processing are stored in PLC. As a PLC address, the hashing value of the session information is used to analyze the spatial locality of packets. However, if a later packet arrives in PLC while an earlier packet is still being processed by a PU in the same session, the later packet does not match the cache entry in PLC, so the packet will be processed by another PU. To reduce this inefficient allocation of PU, we implemented a Cache Miss Table (CMT) and Cache Miss Queue (CMQ) in CMH. CMT stores a packet that has been processed by PU. When a later packet arrives in CMH, CMH checks whether CMT has the table entry for this packet. If the packet matches a table entry, the packet is sent to a queue called CMQ. After the earlier packet in the same session is processed by PU, the later packet in CMQ is processed based on the processing results for the earlier packet without being allocated to PU. Thus, the total cache-hit ratio is the summation of the hit ratio in PLC and CMH, while the cache-miss ratio for P-Gear is the ratio of packets allocated to PU.

In previous research, we concluded that four methods for associativity and a ca. 4K entries cache were an appropriate structure of PLC. It was also appropriate to use ca. 1K entries CMT and ca. 16K entries CMQ, respectively, according to the scaling limitation. We now propose a method for increasing the cache-hit ratio by adding a small hardware or implementing a tiny routine into the processing units of P-Gear.

4 Multi-Context-Aware Cache

P-Gear can utilize the temporal locality of packets effectively and it aims to achieve high throughput using a small number of PUs. In this section, we propose and describe Multi-Context-Aware Cache, which extends the cache mechanism of P-Gear to exploit the PLC cache more efficiently.

P-Gear captures the results of a process during one session into a PLC cache entry. When a cache hit occurs, P-Gear can process the packets of a stream using special wired logic. However, all the packets are cached to PLC with same priority. Because there are other trends of network traffic in addition to the temporal locality, we consider that it is effective for PLC to cache packets with different priorities instead of a uniform priority. For example, if a large number of packets with various headers are concentrated into P-Gear in a short period of time, degradation occurs in the cache-hit ratio because PLC stores useless entries. To prevent these useless entries, PLC is required to determine whether an incoming entry is reused based on implicit information in the IP layer and application layer. This context-based cache management is required to improve the efficiency of processing. Here, we propose a Multi-Context-Aware Cache, which facilitates controlled insertion and deletion of the cache entry by using several contexts simultaneously to achieve a higher cache-hit ratio without increasing the size of cache entries.

It is possible to use the specification of the services of an application layer as a context to determine whether the first packet from a service follows succeeding packets. For example, File Transfer Protocol (FTP) has a specification. It uses a control port and a data transfer port for communication. When a packet arrives at PLC and the source or destination port indicates that the packet is an FTP control packet, PLC can predict that it should not cache the control packet but that it should make a another cache entry for following data packets. This is called an FTP-Aware Cache. As another

example, the Session Initiation Protocol (SIP) of the Voice-of-IP Protocol also has the same behavior and this is known as an SIP-Aware Cache. These cache architectures were proposed and evaluated in our previous studies, and these architectures improved the cache-hit ratio by several percent when used in an IP backbone router [7]. In this paper, we propose a DNS-Aware cache: Rare-DstPort-Aware Cache and P2P-Aware Cache as a new cache architecture based on this strategy.

It is also possible to use request-and-reply transactions as a context, which is generally used in network services. The headers of reply packets can be predicted by interchanging the source and destination information of a request packet, which will be effective for making a cache entry that is generated according to this rule beforehand. We propose a Look-Ahead Cache based on this strategy.

Furthermore, it is also possible to use the characteristics as a context in a network attack. For example, when the packets of a port scan attack are concentrated in P-Gear, P-Gear generates cache entries for a large number of packets that have different destination IP addresses or ports in a short period. These entries will not be reused and they disturb the registration of PLC entries that will be used. Because the packets in a network attack are sent from a source to various destinations and they will not be sent to the same address again, it is effective for the cache not to make entries for packets that are generated by a source node and that are distributed to many different destinations. Similarly, attacks from a worm and DoS attacks often use specific source or destination ports, so it would be effective not to store packets with specific port numbers. We propose an Attack-Aware Cache based on these strategies.

In this paper, we propose and evaluate DNS-Aware Cache, Rare-DstPort-Aware Cache, P2P-Aware Cache, Look-Ahead Cache, and Attack-Aware Cache as subsets of Multi-Context-Aware Cache.

4.1 DNS-Aware Cache

Sessions on a Domain Name System (DNS) have no temporal locality because DNS communication is composed of one request packet and one reply packet, while the session is composed of only one packet. DNS transactions are cached by domain name server and they are rarely reused. Thus, we propose a DNS-Aware Cache, which is a mechanism that does not store the packets from DNS port #53.

We analyzed network traffic to find the ratio of DNS sessions traffic. The analysis showed that over 50% of all sessions were composed of only one packet. These types of sessions are known as one packet sessions. Table 1 and Table 2 show the frequent port numbers and the percentages of >1 packet sessions and one packet sessions. Packets from port #53 mainly appeared in one packet sessions when comparing

TABLE I
Rate of frequent ports in >1 packet sessions

Port number	Source		Destination	
	Quantity	Rate[%]	Quantity	Rate[%]
53	89,998	4.9	149,415	8.1
80	267,924	15	292,905	16
445	169,822	9.2	3,969	0.22
1433	430	0.023	211	0.011
6000	0	0	369	0.020
6881	17,398	0.94	16,725	0.91

TABLE II
Rate of frequent ports in one packet sessions

Port number	Source		Destination	
	Quantity	Rate[%]	Quantity	Rate[%]
53	1,649,237	36	1,700,323	37
80	8,642	0.19	54,057	1.2
445	10,022	0.22	170	0.0037
1433	198,260	4.3	2,200	0.048
6000	5,348	0.12	424,482	9.2
6881	39,249	0.85	37,581	0.81

Table 1 and Table 2. Furthermore, Table 2 shows that 73%of one packet sessions were DNS packets. Therefore, DNS-Aware Cache allows us to reject most of the one packet sessions and to utilize the cache entry more effectively.

4.2 Rare-DstPort-Aware Cache

Sessions with rarely used destination ports have no temporal locality, so there is no need to store them. We propose a Rare-DstPort-Aware Cache, which finds rarely used destination ports in a time window and does not store them in the next time window. The mechanism is shown below.

1. We set a window size and threshold, while a counter counts how many packets with the same destination port arrive in the window per destination port.

2. When the counter reaches the window size, Rare-DstPort-Aware Cache determines the destination ports that are not over threshold as rare destination ports.

3. In the next window, Rare-DstPort-Aware Cache does not store the packets with rare destination ports.

4.3 P2P-Aware Cache

A P2P protocol such as BitTorrent connects with extremely many nodes and the communication of the P2P protocol often finishes with only one packet. These situations will cause a disturbance in the cache. For this reason, we propose a cache mechanism that does not store packets with the specific ports used in the P2P protocol. This is known as a P2P-Aware Cache. It is known that BitTorrent uses 6881 to 6999 as a destination port and it mainly uses 6881. However, some P2P sessions are composed of >1 packets. Because of

this, the P2P-Aware Cache determines whether a BitTorrent session is finished by a packet by utilizing the packet length values of the packet headers, so it does not create a new cache entry if the packet lengths of the packets are less than certain values.

4.4 Look-Ahead Cache

When a packet arrives in a router, reply packets will often arrive soon after, while the difference between the arriving and replying packet header shows the source and destination IP and port that are exchanged. We propose a Look-Ahead Cache, which is a mechanism that creates a cache entry for reply packets based on the incoming packets by exchanging the source and destination information in advance. Figure 2 shows the architecture of the Look-Ahead Cache. In this figure, a packet arrives at the router, which comes from line card A and goes to line card B. At the ingress of Line card A, a cache entry of its egress is created at the same time during the ingress processing. This information is forwarded to the Line card B. Line card B creates a cache entry based on the header information of the outgoing packet or the forwarded information from Line Card A. The cache entry of egress of Line card B is also created at the same time. The ingress and egress network processor in line card A and B can process the concerning packet without allocating a PU.

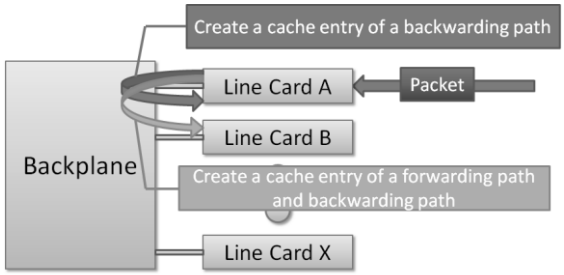


Fig. 2 Architecture of Look-Ahead Cache

4.5 Attack-Aware Cache

As described earlier, a port scan attack disturbs effective cache entries because P-Gear tries to store port scan packets with different destination ports. Therefore, it causes degradation of the cache-hit ratio. To reduce this degradation, we propose an Attack-Aware Cache that identifies network attack packets in network traffic and does not create a cache entry for these packets. In this paper, we propose two dynamic methods and one static method for determining network attacks.

Dynamic methods focus on the fact some types of attacks often send a large number of packets from a specific source node to a large number of destination nodes. To determine whether attackes are waged or not, it is effective to analyze the trend of traffic in increments of a certain amount of network transactions. If the number of connections exceeds a threshold in a window, a new entry is not created in the following window. We set the window size to 512 packets and the method applies results for each window to the next window to prevent entry creation. Dynamic method 1 processes network streams by two processing phases. In phase 1, a dominant source IP at the point of the number of receving packets is marked. Figure 3 shows the mechanism of this method. When a packet arrives, PLC calculates the hashed value of its source IP address using CRC. A 7-bit value is used as the hashed value. Next, A 5-bit counters count the number of packets arriving with the same hash value. If the counter causes an overflow, the IP addresses group that has the same hashed value is a candidate of dominant source in this window. In phase 2, the group is determined whether it conteins an attack source or not. Figure 4 shows the mechanism of this method. The method uses a memory that has three bits addresses. When a packet that has the dominant source arrives, 3-bit counters are incremented accrding to the destination address. As this counters, the 3-bit hash value of its destination IP address is used as an address in the memory. If the number of the counters which is not zero exceeds a threshold exist, this source IP group of the same hashed value is regarded as an attack source. PLC does not cache the packet from this IP group in the following window. Next, we describe dynamic method 2. Figure 5 shows the mechanism for this method. This method generates a hashed value for the source and destination IP address based on the arriving packet respectively using CRC. It calculates a 7-bit value as the

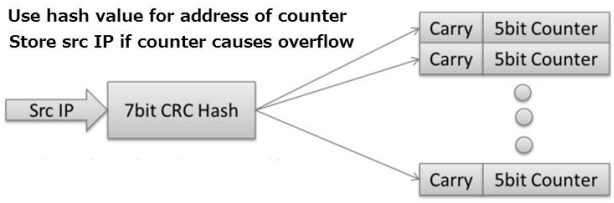


Fig. 3 Mechanism of dynamic method 1 in phase 1

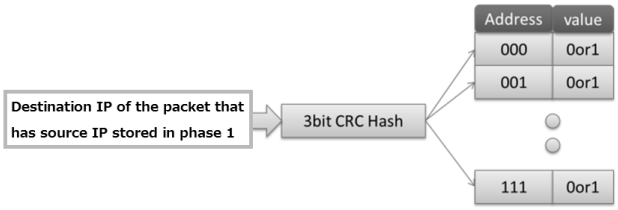


Fig. 4 Mechanism of dynamic method 1 in phase 2

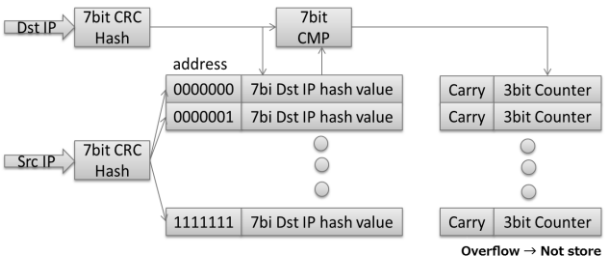


Fig. 5 Mechanism of dynamic method 2

hashed value and it stores the hash value of the destination IP address in memory. At this point, it uses the hash value of the source IP address as an address in its memory. When the same address is accessed again, it compares the hash values with the destination IP address. If the values are different, it increments a 3-bit counter. If the number of the counter exceeds a threshold, a new cache entry is not created until the counter is reset.

Static methods focus on the fact that some types of attacks use specific ports. For example, DoS attacks exploit the vulnerability of particular applications such as database services, so the attack packets used the specific ports for database services. Furthermore, the headers of these attack packets are often very variable. We analyzed the network traffic to find the ports used by attacks, and Table 1 and 2 show the results. We found that packets with destination port #1433 and source port #6000 were sent in bulk during one packet sessions. Packets with the destination port #1433 were sent to SQL as worm attacks, while packets with source port #6000 were sent as port scan attacks. By analyzing the network traffic, we found that packets with source port #1434 and source port #3306 could also be regarded as network attack packets. The static method identifies packets using these ports as attack packets and it does not create cache entries for them.

5 Evaluation

In this section, we show the simulation results, where we evaluate the five proposed cache mechanisms. We used P-Gear simulator and WIDE trace (150Mbps) as real network traffic for this evaluation. P-Gear simulator was written in C++ while PLC was implemented as a set associative cache using four methods. The size of cache entries was 1,024 because we considered a 394-bit PLC tag and 365-bit PLC data, so the capacity of the cache memory was estimated as 89 Kbytes. This size of the cache can be implemented as a high-speed on-chip L1 or L2 cache. The number of PUs was 32 and a delay when processing a packet was 2 μs in the simulations.

5.1 Results with the DNS-Aware Cache

We simulated the DNS-Aware Cache and compared it with a normal cache mechanism that caches all ports. Figure 6 shows the cache-miss ratio with the normal mechanism and the proposed mechanism over 12 hours (from 0:00AM to 12:00AM). The cache-miss ratio with DNS-Aware Cache was reduced by roughly 1-2 points, which led to a ca. 6% improvement in the packet processing costs of the PU. The DNS-Aware Cache eliminated most of the one packet sessions by preventing the creation of new entries for DNS packets, which was effective for reducing the cost of packet processing.

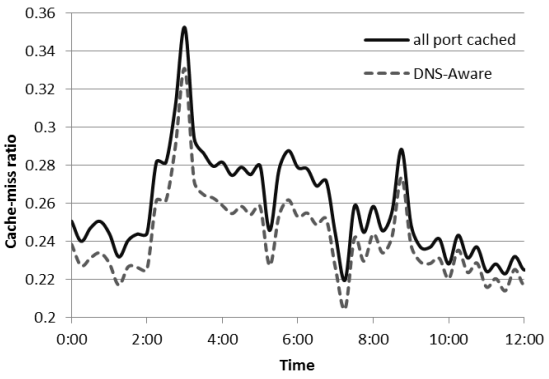


Fig. 6 Simulation results for the Cache-miss ratio with the DNS-Aware Cache

5.2 Results with the Rare-DstPort-Aware

Figure 7 shows the simulation results for the cache-miss ratio with each window size. We changed the window size (x-axis) and the threshold n for the Rare-DstPort-Aware Cache, as shown in the different graphs. With all thresholds, the cache-miss ratio was improved by increasing the window size. However, the cache-miss ratio with this mechanism was never less than the ratio when all destination ports were cached. This was because the destination port identified in the previous window appeared in the next window in some cases. When the window size was 5,000,000 packets, the number of unique destination ports was 8,426. However, one of these destination ports appeared in next window, which led to an increase in the cache-miss ratio. Furthermore, caching destination ports that never appeared in the window was sometimes ineffective because they may also appear in next window. We needed to consider a method that could identify rare destination ports more accurately.

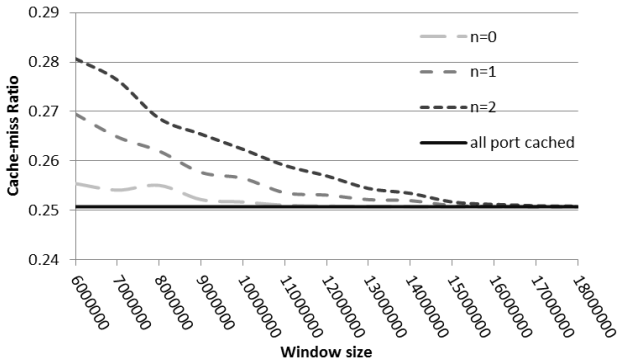


Fig. 7 Simulation results for the Cache-miss ratio with the Rare-DstPort-Aware Cache for each window size

5.3 Results with the P2P-Aware Cache

We simulated the P2P-Aware Cache using the same traffic and we compared the four mechanisms shown below.

A) Caching all ports (all ports cached)

B) Not caching port 6881 only, as mainly used by BitTorrent (P2P_6881)

C) Not caching port 6881 to port 6999, as used by BitTorrent (P2P_6881-6999)

D) Not caching port 6881 and less than 256-byte length packets (P2P_P-Length)

By analyzing packet length values of the packet headers, mechanism D determined whether the P2P sessions consisted of one packet or more. The threshold of the packet length was 256 bytes. If the packet length value was over the threshold, we identified the packet as a session with one packet.

We compared the cache-miss ratio with the P2P-Aware Cache and the cache-miss ratio using the all ports cached mechanism, and Figure 8 shows the results. All the P2P-Aware mechanisms improved the cache-miss ratio in many cases. However, P2P_6881 and P2P_6881-6999 sometimes increased the cache-miss ratio. This was because these mechanisms did not create entries for sessions composed of >1 packets, although packets for these sessions appeared in some cases. However, with P2P_P-Lngth, this increase in the cache-miss ratio was less than one-third, while the cache-miss ratio was improved by about 0.5%.

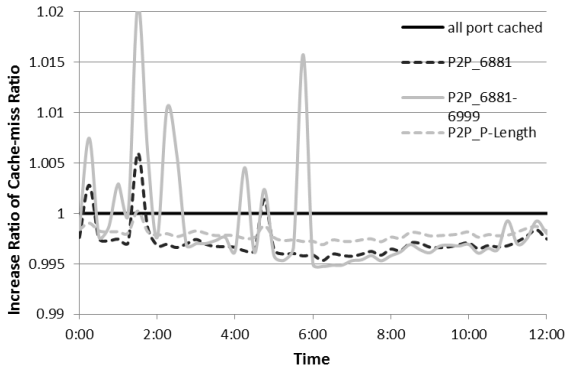


Fig. 8 Simulation results showing the increased cache-miss ratio with the P2P-Aware Cache

5.4 Results with the Look-Ahead Cache

We simulated the Look-Ahead Cache using the same traffic. This simulation assumed that request-and-reply transactions were processed on the same line card. Figure 9 shows the cache-miss ratio in the simulation. This shows that the mechanism reduced the cache-miss ratio by 2% compared with the normal mechanism while the cache-miss ratio using this mechanism was always lower than the normal mechanism. This demonstrates that the Look-Ahead Cache was effective and it improved the packet processing costs by about 9%.

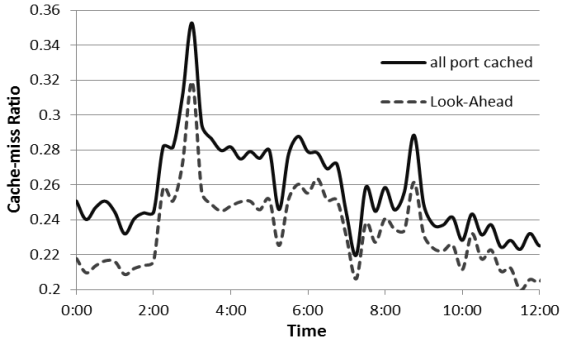


Fig. 9 Simulation results for the cache-miss ratio with the Look-Ahead Cache

5.5 Results with the Attack-Aware Cache

We simulated two dynamic mechanisms and one static mechanism with the Attack-Aware Cache. In the simulations of dynamic mechanisms, we changed the threshold from 4 to 32. Figure 10 and Figure 11 show the results of simulations for dynamic mechanisms each second. If the cache-miss ratio increased suddenly, both mechanisms reduced the cache-miss ratio by about 2 points. However, they increased the average of the cache-miss ratio because the source port determined by the dynamic mechanisms included sessions composed of >1 packets. As shown in figure 10 and Figure 11, dynamic mechanism 2 was more effective than dynamic mechanism 1.

In the simulations with the static mechanism, the mechanism did not cache packets that included destination port #1433, #1434, and #3306, and source port #6000. Figure 12 shows the shift in the cache-miss ratio during each second. As shown in Figure 12, when the cache-miss ratio increased suddenly, the static mechanism reduced the cache-miss ratio by 4%. The mechanism did not degrade the average cache-miss ratio in many cases. Therefore, this static mechanism was effective during network attacks and improved the packet processing costs by about 10% when the cache-miss ratio increased suddenly due to network attacks.

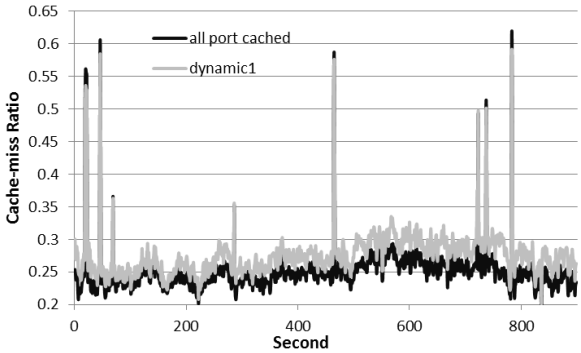


Fig. 10 Simulation results for the cache-miss ratio using the dynamic Attack-Aware mechanism 1

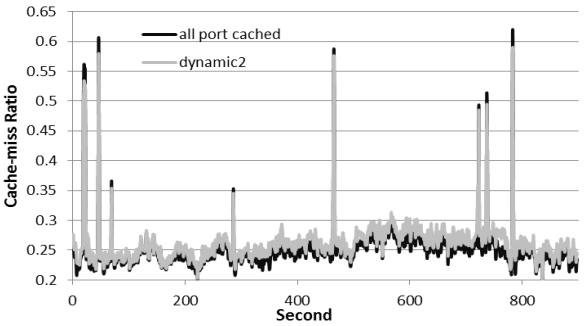


Fig. 11 Simulation results for the cache-miss ratio using the dynamic Attack-Aware mechanism 2

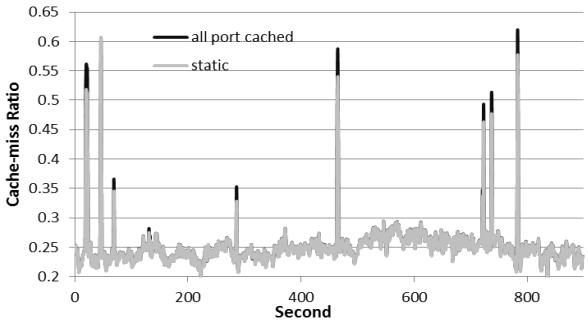


Fig. 12 Simulation results for the cache-miss ratio using the static Attack-Aware mechanism

6 Conclusions

We proposed P-Gear, which has a special cache architecture that efficiently utilizes the temporal locality of network traffic to achieve a higher throughput with a limited number of PUs. For P-Gear, the reduction of the cache-miss ratio is a significant requirement for reducing the processing costs of PUs. We proposed a cache mechanism known as Multi-Context-Aware Cache to improve the cache-miss ratio with P-Gear. Multi-Context-Aware Cache estimates trends in network traffic by continuously analyzing the packet headers and it optimally manages cache entry.

In this paper, we proposed and simulated five mechanisms that formed a subset of Multi-Context-Aware Cache. DNS-Aware Cache, Rare-DstPort-Aware Cache, and P2P-Aware Cache utilized port information to determine whether sessions had temporal locality. Look-Ahead Cache utilized the relationships of request-and-reply transactions. Attack-Aware Cache utilizes the trend in network attacks. In simulations, the DNS-Aware Cache eliminated about 70% of a session composed of one packets and it improved the cache-miss ratio by 6%. However, it was difficult to estimate rare destination ports using Rare-DstPort-Aware Cache. This method did not improve the cache-miss ratio, so there is still a need to estimate rarely used ports more accurately. P2P-Aware Cache eliminated the packets of BitTorrent, which often communicates using one packets and it improved the cache-hit ratio by 0.5%. However, we need to find a more precise method for determining whether sessions of packets are composed of only one packet. Furthermore, the Look-Ahead Cache estimated the reply packet headers and improved the cache-hit ratio by 9%. Finally, the Attack-Aware Cache identified network attack packets and improved the cache-hit ratio by almost 10% when the cache-miss ratio increased suddenly.

7 Acknowledgment

This work was partially supported by National Institute of Information and Communications Technology (NICT) and a Grant-in-Aid for Scientific Research (C) (22500069).

8 References

[1] P. Crowley and M.A. Franklin and H. Hadimioglu and P.Z. Onufryk. "Network Processor Design: Issues and Practices Volume 1". Morgan Kaufmann, 2002.

[2] Douglas E. Comer. "Network Systems Design using Network Processors (IXP1200Version)". Pearson Prentice Hall, 2004.

[3] Tzi Cker Chiueh and Prashant Pradhan. "High-Performance IP Routing TableLookup Using CPU Caching"; In Proceedings of INFOCOM'99, pp. 1421—1428, 1999.

[4] Tzi Cker Chiueh and Prashant Pradhan. "Cache Memory Design for Network Processors"; In Proceedings of IEEE High Performance Computer Architecture Conference (HPCA) 2000, pp. 409—419, Jan. 2000.

[5] Bryan Talbot, Timothy Sherwood, and Bill Lin. "IP Caching for Terabit SpeedRouters"; In Proceedings of Global Communication Conference (Globecom'99), pp.1565—1569, 1999.

[6] Michitaka Okuno and Hiroaki Nishi. "Network-Processor Acceleration-Architecture Using Header-Learning Cache and Cache-Miss Handler"; In The 8th World Multi-Conference on Systemics, Cybernetics and Informatics, SCI2004, Vol. III, pp. 108—113, Jul. 2004.

[7] A. Akimura and H. Nishi. "Multi-context-aware cache accelerating processing on network processors for future internet traffic"; In Advanced Communication Technology (ICACT), 2010 The 12th International Conference on, Vol. 1, pp. 377—382.IEEE, 2010.

[8] K. Gopalan and T.C. Chiueh. "Optimizations for Network Processor Cache"; In Proceedings of SC2002 High Performance Networking and Computing, 2002.

[9] I.L. Chvets and M.H. MacGregor. "Multi-zone Caches for Accelerating IP Routing Table Lookups"; In High Performance Switching and Routing (HPSR) 2002, pp.121—126, 2002.

Evolution of the Internet Autonomous System Network's Topological Pattern

Craig Stewart and Javed I. Khan
Department of Computer Science
Kent State University, Ohio, United States
cstewartl javed @ cs.kent.edu

***Abstract*—How the topology of Internet is evolving? In this research we present a large scale longitudinal analysis of the changing topological pattern of Internet Autonomous Systems Network (ASN). Analysis of topological patterns for large scale network is not trivial. We introduce a novel technique called *generating function signatures* (GFS). It finds statistical signatures of various pattern graphs and observes the changes in snapshots of ASN collected over a period of five years. The study reveals several interesting trends about the evolutionary state of the ASN including its increasing trellis-like topological evolution.**

Keywords- topological analysis; generating function; internet.

I. INTRODUCTION

The Internet is a network of *autonomous systems* (AS). Each AS itself is a network –operated as one administrative domain owned by regional and national ISPs, large organizations, and customers. Autonomous systems are the basis of the distributed management of the Internet. Long haul Internet routing is performed on the basis of autonomous systems [1].

A number of researchers have studied various properties of on the ASN [2,3,4,5,6], such as size, degree distribution, clustering etc. However, very little previous studies exist on the evolution of its topological structure. This is because, the topological property study of the autonomous systems poses many challenges, making it difficult to examine and monitor them. First of all, the size of the ASN is very large. Fig- 1 shows the ASN in March 2004. The numbers of autonomous systems were over 7000 and the peer connections between them were 15000 about ten years ago. Since, then both numbers are increasing rapidly – at times reported to be exponentially [2,7] – on an annual basis. Secondly, there is no ideal pattern in such scale. The very concept of any concrete pattern degenerates in large scale systems making it hard, if not impossible, to apply any deterministic graph traversal technique. As evident in Fig-1 the network is complex- it has few recognizable topological features; the highly-connected nodes are in the center and the lower degree nodes branch out into points similar to a star topology. However, such visual recognition of structure has become impossible over time. The ASN also exhibits a steady level of dynamism. The network is in a constant state of evolution.

Despite the difficulties, nevertheless the analysis of the topological structure of the ASN can provide fundamental insight about the requirements and design on next generation internet. Oliveira et al [8] writes: "It [the Internet topology] provides an essential input to the understanding of limitations of existing routing protocols, the evaluations of new designs, as well as the projection of future needs; and it will help advance our understanding of the interplay between networking technology, the resulting topology, and the economic forces behind them". Lee and Kalb [9] from Sandia Lab details strategic advantages those can be derived from various network topologies ranging from familiar patterns like stars and rings to complex three- and four-dimensional structures like cubes and toroids in large scale communication networks. Kit is well known that knowledge about topological structure can help improving efficiency of systems [10, 11, 12].

In this paper we introduce a new technique based on *generating functions (GF)* that can overcome few of the difficulties. In this study we demonstrate a general process detailing the ways generating function signatures (GFS) can be used to trace the presence of various topological patterns in a large scale network by searching for the signature of specific topological patterns. These signatures are conceived to be a statistical characterization for each of these structures. With the aid of such GFS we are able to undertake one of the most comprehensive studies of the structural evolution for the ASN.

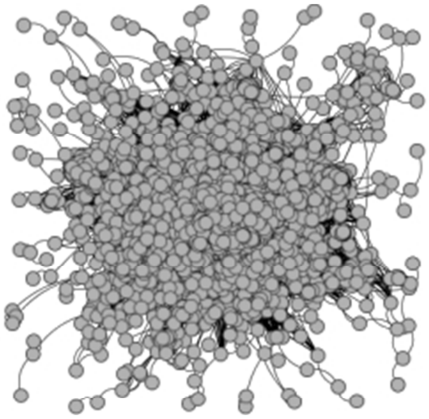


Fig-1 A map of the topology of the ASN in March 2004.

The data set used was obtained from Oliveira et al [13] for the analysis. This extensive trace of the ASN is built from data available from various AS observation sites such as BGP route tables and updates collected from numerous large networks, including RouteViews, RIPE-RIS, Abilene, CERNET route servers at Internet Exchange, BGP View, Packet Clearing

House, UCR, traceroute.org, Route Server Wiki, and Looking Glasses like NANOG and the Looking Glass Wiki. We present our findings on this paper.

The paper is organized in the following way. Section 2 covers recent research related to this work. Sections 3 and 4 then explain the process by which generating functions can be used for structural property analysis in a complex graph. We explain four specific topological patterns before discussing the method of searching for signatures of such patterns in a large complex network such as the ASN, including networks that mix multiple patterns. Finally, Section 5 presents the analysis and observations of the ASN as it evolved during the years 2004-2009.

II. Related Works

One of the earliest studies of ASN is attributed to Magoni and Pansiot [7] which analyzed six snapshots of ASN from November 1997 to May 2000 based solely on BGP data [7] and analyzed its size growth. Vazquez et al [2] used the autonomous system map from May 2001 and also took measures of the clustering coefficient and average betweenness of the nodes. Among other notable works are [3, 6] which found progressively more ASNs. Clegg et al [4] seems yto be the first to bring attention to topological analysis and proposed a 'Framework for Evolutionary Topology Analysis' (FETA) for the study of evolutionary topology. FETA provides a fast process comparing a model of a network to its real-world counterpart and taking measurements on its evolutionary qualities. This framework was used to study the ASN from January 2004 to August 2008. Their study revealed information such as degree 1 and degree 2 node count, maximum degree, and clustering based on the size of the network. Clegg et al reported declining degree 1 node counts and increasing degree 2 node counts as the number of connections rose. In addition, the clustering coefficient remained less than 0.06 regardless of network size. None of the studies however could proceed to analyze the topological pattern evolution of such a large ASN. This is because high computational cost of such analysis in complex networks with degenerated patterns.

In this endeavor we have also concluded an extensive longitudinal analysis of various complex network properties of the ASN (such as size, community structure, degree distribution etc) with unprecedented ev0olutionary view of the ASN. Interested researcher might want to read [14]. This paper particularly focuses on the evolution of topological pattern of ASN. As indicated before the proposed analysis is based on a novel use of generating functions (GF). Although GFs have not been used in such a way before, but GF techniques have been very useful in the analysis of idealized random graphs. Newman et al [15] introduced degree sequences in the form of generating functions and then derived various composite properties of graphs with specific distribution. Callaway et al [16] used generating functions in order to find analytical solutions to site percolation in random graphs and provide a process for testing the robustness of a network to the random failure of nodes and links by manipulating the generating functions of their random graphs. Generating function has also used to study directed ideal network graphs. Dorogovtsev and Mendes in [17] involved generating functions (called Z-transforms in their paper) for the degree distributions of networks in a directed graph that show preferential linking to solve Masters equations and demonstrated an efficient method for finding solutions to network evolution problems. Dorogovtsev and Mendes also developed a Master equation that represents the in-degree distribution of a network over multiple iterations as a Poisson generating function. It was based on the distribution of nodes, number of edges added to the network, and additional attractiveness of nodes to receive those new edges.

In this paper we use GFs as a signature of various topological pattern families and try to find topological resemblance in ASNs.

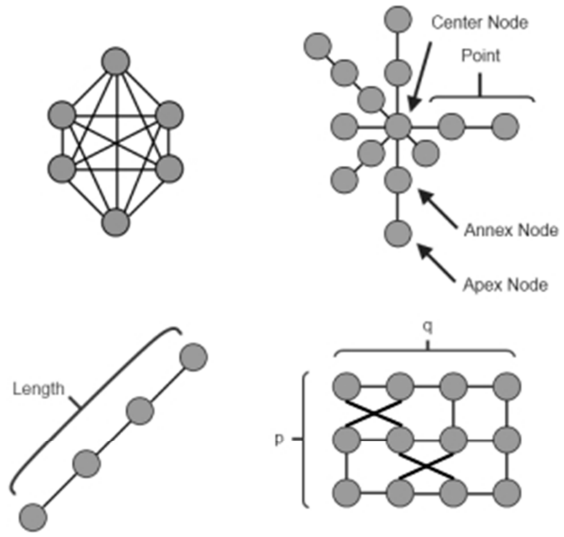


Fig-2 Examples of mesh, long star, chain and trellis (clockwise from top).

III. Special Pattern Graphs

The method of topological analysis proposed in this paper makes use of ordinary generating functions (OGF). A generating function is a polynomial function that represents some aspect of a network. They most commonly contain the degree distribution information for the nodes in a specific network. The ordinary form is used, similar to the form used by Newman et al in [15]:

$$G_0(x) = \sum_{k=0}^{\infty} \quad p_k x^k. \qquad (1)$$

Each k value represents the degree while the pk coefficients hold the probability that one node out of all nodes in the network has degree k. For example, the generating function G0(x) = 0.5x2 + 0.5x contains the information for a network that has an even distribution of degree 1 and degree 2 nodes.

Generating functions are used in this paper as a means of identification of various patterns in the network. The generating function of a network is compared to the generating function for special patterns to determine their likeness to each other.

A pattern graph is a regular topological structure. Fig-2 shows examples of four such patterns. Equation (2) is the GF of a *fully connected mesh* pattern when it has n nodes.

$$G_{mesh(n)}(x) = x^{n-1}. \quad (2)$$

A *star*, as it is commonly known, is a structure of *n* nodes similar to a tree with the central node being degree (*n*-1) and the other *n*-1 nodes being degree 1. For the purposes of this paper, we consider a generalized star with long legs referred to as *long star*. A long star has nodes of three types: the center, the apexes, and the annexes. The center node in the star has the highest degree *d* from which the points branch off. A long star pattern with n nodes and dimension *d* has generating function:

$$G_{s(n,d)}(x) = \frac{1}{n}x^d + \frac{n-d-1}{n}x^2 + \frac{d}{n}x. \quad (3)$$

A *chain* structure of *n* has generating function:

$$G_{c(n)}(x) = \frac{n-2}{n}x^2 + \frac{2}{n}x. \quad (4)$$

The *trellis mesh* (or simply just *trellis*) is a system of nodes arranged like a square matrix defined by a height and width *p* x *q*, requiring that $p \geq 2$ and $q \geq 2$. All nodes in this pattern have a degree of 2, 3, or 4 based on their position. One generating function for a trellis provides for many combinations of squares and crosses formed by the edges between each set of nodes. Fig-2 shows an illustration of a 3x4 trellis. A trellis structure with dimensions p x q has generating function

$$G_{g(p,q)}(x) = \frac{(p-2)(q-2)}{pq}x^4 + \frac{2(p+q-4)}{pq}x^3 + \frac{4}{pq}x^2. \quad (5)$$

IV. Comparison & Similarity Analysis

The analysis process begins by analyzing- (even visually if needed) the shape of the graph, utilizing it to discover its likeness to certain patterns. Network and pattern pairs should look similar to each other; they must share a similar signature, a common topological feature, with one another such as points of a star or a region of high clustering like a trellis. The terms in their generating functions also need to be comparable. Exponent in the pattern must also exist in the network.

Once the patterns have been decided upon for analysis, the *network generating function* (the function for the degree distribution of the network) and the *generating function signature* (the signatures such as (2)-(5) that characterize the generating functions for all patterns within an ensemble using parameters) need to be compared in order to find the right dimensions of the pattern graph.

Comparisons are always done using the *dominant coefficients*, the highest or most significant values from the generating function signature $G_0(x) = \sum f(n,m)$. Consider a generating function signature $G_0(x) = \frac{n-m-c}{n}x^k + \frac{m}{n}x^{k-1} + \frac{c}{n}x^{k-2}$ where *n* is the total node count, *m* is a distinct quality of the pattern that directly contributes to the node count, and *c* and *k* are constants. If the network is very large, the function becomes $\lim_{n\to\infty} G_0(x) \approx r_1x^k + r_2x^{k-1}$ where r_1 and r_2 are fractions of the remaining terms. While the degree (*k*-2) term disappears, the other coefficients remain large and are considered dominants.

With the dominant coefficients decided upon, the parameters for the generating function signature need to be calculated. The ratios of dominant coefficients are used in situations involving multiple dominant coefficients. The parameters from the dominant coefficients are then substituted into the generating function signature to obtain the specific *pattern generating function* (the function for the pattern graph closest to the network.) The error between the network generating function and pattern generating function is then minimized. The error value, E, can be calculated as:

$$E = \sum_{k\in f} | a_k - f_k\Delta |. \quad (6)$$

Here a_k is the coefficient for the degree *k* part of the network generating function, f_k is the coefficient for the degree *k* part of the pattern generating function, and Δ is the value that must be calculated to produce the smallest possible value for E. The L^1-norm is used here since a median of the limited terms is sufficient and for ease of calculation. The inversion of Δ produces a percentage that represents the resemblance between the network and its matching pattern.

A. Chain Similarity

The GF signature of chain (Eq-4) only contains x and x^2 terms and the x term has a dominant coefficient. The only dominant value here is the x^2 coefficient, and it must be set equal to the x^2 coefficient in the network generating function. Then the number of nodes *n* can be found by solving the equation:

$$\frac{n-2}{n} = a_2 \text{ or } n = \frac{2}{(1-a_2)}. \quad (7)$$

The value of a_2 is the x^2 coefficient in the network generating function. Replace *n* in the generating function signature with the value calculated from (7) to obtain the pattern generating function. Finally minimize the error between the two sets of coefficients for the x and x^2 terms and invert the Δ-value to find the chain similarity of the network.

B. Star Similarity

The GF signature of chain (Eq-3) has three terms; however, the x^d coefficient is $1/n$ so it is deemed non-dominant. The x^2 and x terms are dominant since it relies heavily on the node count. The coefficient of the x^2 term is the largest, and the coefficient for the x term is then the second largest. As stated earlier, the ratio of the most dominant and the second most dominant coefficients are taken when two dominant coefficients exist. The node count can then be decided by solving for *n* where a_2 and a_1 are coefficients of x^2 and x.

$$n = \frac{a_2 d}{a_1} + d + 1 \quad (8)$$

Any fractional part of the *n* value can be removed from the result of (8) to form the (*d*,*n*)-pair. These values are then used to create the pattern generating function by substituting them into the generating function signature.

C. Trellis Similarity

The x^3 and x^4 parts of (5) are the most dominant in the trellis structure as they are based on the dimensions of the trellis. The x^2 term, again being a constant value over the node count, does not carry anything significant. The ratio is then the x^4 to x^3 coefficients:

$$\frac{(p-2)(q-2)}{2(p+q-4)} = \frac{a_4}{a_3} \quad (9)$$

The a_4 and a_3 variables represent the x^4 and x^3 coefficients of the network generating function.

In order to obtain the p and q parameter values, some error testing is necessary. A value for q within the range $2 \geq q \geq$ diameter of the network must be found that produces the lowest error from the nearest integer. In other words, the (p,q)-pair that creates the best fit for (9) is needed. The following equation can be used for error testing:

$$p = \frac{2qr+2q-8r-4}{q-2-2r} \text{ for} r = \frac{a_4}{a_3}. \quad (10)$$

It is advised to remain within a threshold of ±0.1 when determining proper values for p. Since both p and q are equally significant in these functions and their values need to be integers, it is important to find the values that best fit the dimensions of the dominant trellis in the network. The error testing is the same as the other structures.

D. Similarity Analysis with Multiple Patterns

Analysis of network is non-trivial when there are multiple patterns present in the network. The simplest case of compounding is when the network is built from multiple occurrences of *congruent* pattern graphs (all with exact same generating function). It can be called *homogeneous forest of congruent patterns.* Complex networks may also be made of non-congruent patterns. These can be called *homogeneous forests*. The generating functions for two non-congruent patterns will differ in at least one of the coefficients or parameters. Further networks can be made of completely dissimilar patterns. These can be called *heterogeneous forest.* The pattern graphs in the forest may be disjoint from each other or linked together by relatively small number of additional edges in some manner. If the pattern features are significant, the small interconnection links may not disturb the analysis in the limit. The GF for complex networks can be stated as:

$$G_{h(n)\in F}(x) = \sum a(n).h(n). \quad (11)$$

This formula represents a network with a set of patterns, F, each basis with its own generating function $h(n)$. Here $a(n)$ is the proportion of nodes attributed to n^{th} pattern.

The GF of homogeneous forest of congruent patterns holds the same ratios of nodes of different degrees in the network like the pattern network. It causes no change in the pattern generating function in the limit. The analysis can proceed in the same way as that of the basic pattern signature analysis. Based on the node counts in basis pattern, the repetition factor can also be estimated.

The analyses of other type of forests are more complex and do not have guaranteed solvability. To analyze forests it is necessary to isolate each individual pattern. If a pattern GF has unique exponent (which is normally the case for heterogeneous forests)- then it normally can be isolated. When they share exponents then various constraints equalities can be formed by analyzing the ratio coefficients. The difficulty by which these equations can be solved lend to three different solvability categories for patterns: inseparable, separable, and separable with difficulty.

The approach to separating the network between patterns starts by determining the first pattern to isolate. It may involve visualize inspection. Separable structures should be isolated before patterns that are separable with difficulty. Once a pattern is chosen, and determined to be present significantly all subgraphs of that pattern from the network is removed. All remaining nodes for the shared terms contribute to the terms in the next pattern search.

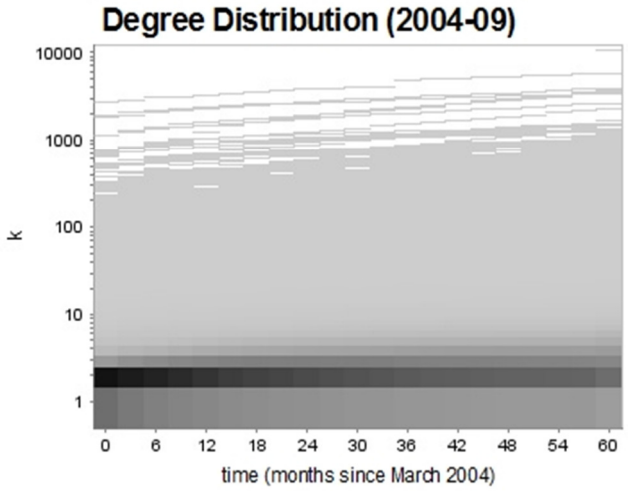


Fig-3. The *evolution of* degree distribution Darker grays and blacks are closer to $p_k = 0.5$ in (a) and $p_k = 0.01$ in (b). White blocks are $p_k = 0$.

Long stars are normally separable, especially when they have high dimension. Each apex of individual long star can be pulled from the network by taking its exponents and determining whether or not they form stars by determining their node counts using the ratio of annex nodes to apex nodes as in (9). On the other hand, the chain is a prime example of an inseparable pattern. It relies only on node count, and the only information held in the coefficients of a pattern generating function for a chain is the number of degree 1 nodes, a constant. With only a constant and an unknown variable, there is no way to solve equations dealing with chains. Trellis, when it is mixed with long-star, is normally separable. However, the non-congruent trellis patterns are only partially separable. Two trellis patterns normally yields under-constraint set of equations.

V. Analysis & Observations

A. Data Set

The extensiveness of the study derives from the massive data set of [13]. It creatively combines data from (a) BGP advertisements and updates observed at various AS observation sites such as large networks including Route Views, RIPE-RIS, Abilene, CERNET, (b) route server data at various Internet Exchanges such as BGP View, Packet Clearing House, UCR, traceroute.org, Route Server Wiki, and (c) data at Looking Glasses such as traceroute.org, NANOG, Looking Glass Wiki. This is the most extensive trace of the ASN available to-date. All data is obtained with publicly available software that allows for the viewing of routing information for the autonomous system of a single entity.

To improve accuracy, the data set did not combine any trace route data since AS paths cannot be accurately inferred from IP paths. The data, however, is still may not perfect. This method still can potentially miss links and nodes [18]. This is due to two reasons- unwillingness of service providers to allow their nodes to be discovered or drop of information in routing tables.

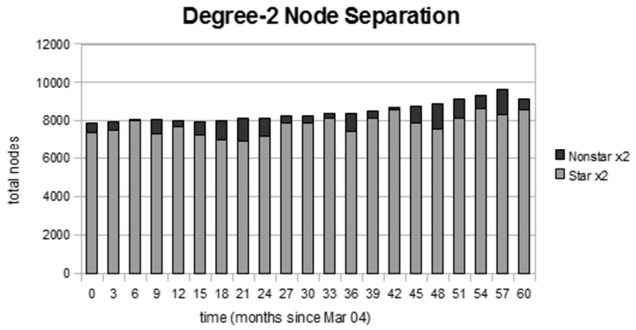
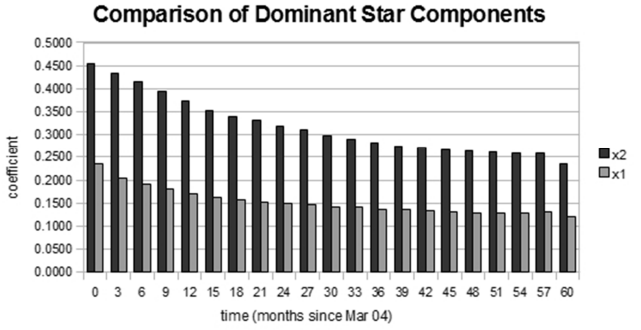
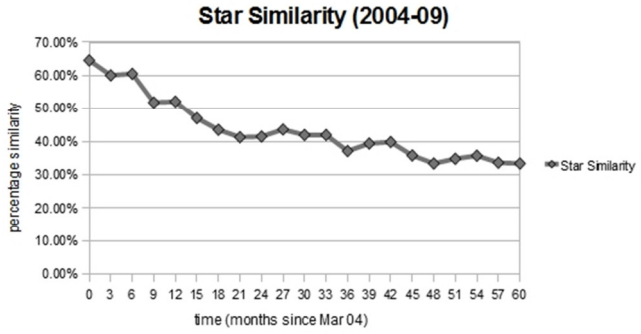


Fig- 4. The separation of the degree 2 nodes in the network between the star and non-star parts prior to analysis.



(a)



(b)

Fig- 5. (a) The distributions of the degree 1 and degree 2 nodes in the autonomous system network. (b) The calculated star similarity, resemblance of the ASN to its corresponding long star for each month of the study.

The data set taken from [13] contains the nodes and links from the first day of every third month for all viable years, ending with the month this project began. A topology map is built each day from these collections. The collected maps span over every March, June, September, and December from March 2004 (the first year the maps were created) to March 2009 (the date this project began.) Although more data is available, these 21 maps are used in the data set for this project.

Each iteration of the network is divided into two files. One carries the list of monitored nodes while the other contains a list of links. The records for these files start with the unique identification numbers of the ASes assigned by the IANA. The node files have one ID and the link files have two IDs, one for each AS connected to that link. The next two fields are the times the node or link was first and last monitored by the software, represented as Unix timestamps (seconds since the epoch.) The last piece of information represents the node or link's position in a customer-provider chain. A 0 is used for the start of a path, 1 for the middle, and 2 for the end.

The degree information and node counts stored for the graph of each month in the study were used to build the generating functions for each one. The functions were written in the form

$$G_0(x) = \sum_{k=1}^{\infty} \frac{t_k}{n} x^k \quad (12)$$

where k is the degree, t_k is the total number of nodes that were of degree k for that month, and n is the node count for the entire network. The generating functions were formulated this way to preserve each piece of data for future use.

B. Degree Distribution Trends Observed

The degree distributions showed some interesting results as illustrated in Fig-3. Each block represents the p_k value, distribution for nodes of degree k, for the corresponding degree and month in the study as indicated by the axes. The degree 1 and degree 2 node counts were the highest. They show up as distinct, dark colored bands that grow lighter over time. This trend shows that the ASN is slowly moving away from the stub configuration. The bands for the degrees between 3 and 10 follow the opposite pattern, starting light and gradually becoming darker as time moves on. These changes indicate that autonomous systems with one or two peers appear to be obtaining more links to providers or peers over time. Going across the chart from month to month, the contrast of the bands varies at all values of k. It can be seen that the darker colored blocks (the distributions of higher p_k) are moving higher over time, causing diagonal bands.

C. Dominant Pattern Analysis

The generating functions, metrics, and visual graphs were used to determine the pattern graphs to be tested. Among the many observations, the ASN shows signatures of the long star and the trellis mesh ensemble as the x, x^2, x^3, and x^4 terms of the generating functions were all fairly high. These two patterns share the x^2 term, however, so the degree 2 nodes would need to be divided between the two patterns.

Fig-4 shows the separation of the degree 2 nodes in the ASN for the long star ensemble. Since the long star is under the separable solvability category, the degree 2 nodes for the long stars were separated from the nodes that did not belong to them. The leftover nodes would be attributed to the trellis mesh ensemble. Due to the difficulty of separation for trellises, each iteration of the network was considered to contain only one trellis when determining their trellis similarity percentages. For

the separation of the star nodes, the system of equations is not used here since the x^d term is most significant in determining the star parameters and too many large degree numbers exist

Table-I Dominant Trellis Dimension over Time

Months since March 2004	Year	p	q
0 - 9	2004	4	5
21-Dec	2004	4	6
24 - 36	2005-06	4	7
39, 42	2006-07	4	8
45, 48	2007-08	4	9
51 - 60	2008-09	4	10

Note: Based on error testing as in Table II.

Table-II Dominant Trellis Error Testing

ratio	q=2	q=3	q=4	q=5	q=6	q=7	q=8	q=9	q=10
0.38	0	11.5	4.5	3.7	3.4	3.3	3.2	3.1	3
0.39	0	12.6	4.6	3.8	3.5	3.3	3.2	3.1	3.1
0.4	0	14	4.7	3.8	3.5	3.3	3.2	3.2	3.1
0.43	0	20.4	5	4	3.6	3.5	3.3	3.3	3.2
0.46	0	36.5	5.4	4.2	3.8	3.6	3.4	3.4	3.3
0.47	0	49	5.5	4.3	3.8	3.6	3.5	3.4	3.3
0.48	0	74	5.7	4.4	3.9	3.7	3.5	3.4	3.4
0.5	0	0	6	4.5	4	3.8	3.6	3.5	3.4
0.51	0	-151	6.2	4.6	4.1	3.8	3.6	3.5	3.5
0.53	0	-51	6.5	4.7	4.2	3.9	3.7	3.6	3.5
0.54	0	-39	6.7	4.8	4.2	3.9	3.8	3.6	3.6
0.55	0	-31	6.9	4.9	4.3	4	3.8	3.7	3.6
0.59	0	-18	7.8	5.2	4.5	4.2	4	3.8	3.7
0.61	0	-15	8.3	5.4	4.6	4.3	4	3.9	3.8
0.63	0	-13	8.8	5.6	4.8	4.4	4.1	4	3.9
0.64	0	-12	9.1	5.7	4.8	4.4	4.2	4	3.9
0.65	0	-11	9.4	5.8	4.9	4.5	4.2	4.1	3.9
0.66	0	-10	9.8	5.9	5	4.5	4.3	4.1	4
0.7	0	-8.5	11.3	6.4	5.2	4.7	4.4	4.3	4.1

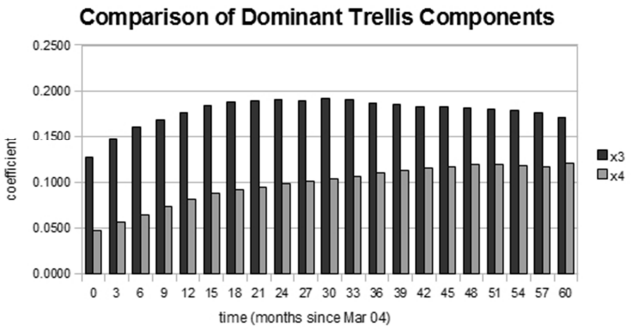
Note: Light gray values represent best fit. Dark gray values are too far out of the range.

for the number of degree 1 nodes. In this case, all degree values less than 5% of the node count were used as potential dimension numbers in order to limit *m*. The dimension values were substituted into (9) and then tested for error from the nearest integer. If the error was under 0.1 then the dimension and node count were kept as part of the pattern GF.

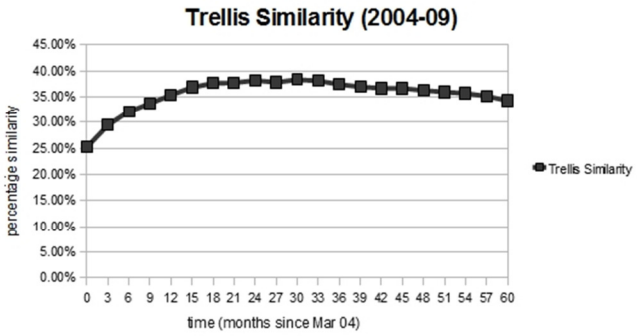
The view of the network as a heterogeneous forest of long stars and one trellis weighs heavily on the separation of degree 2 nodes in the network. Fig-4 shows that the majority of these nodes are part of the long star structures in the network. The trellis pattern has a small constant for the x^2 term in its generating function, so the low numbers help the error testing phase of trellis similarity calculation.

D. Star Ensemble Similarity Analysis

The visualization of the early network graphs (as in Fig- 1) showed significant presence of long star like ensembles at the edges of the network. In Fig- 5a we show the degree measures for significant components of the long star ensemble i.e. the numbers of nodes with degree 2 and degree 1 within the ASN. These were higher than all other numbers and several centers appeared to exist as many of the high degrees had p_k values of $1/n$. These two aspects made finding the star similarity of the ASN favorable.



(a)



(b)

Fig- 6. (a) The distributions of the degree 3 and degree 4 nodes in the ASN. (b) The calculated trellis similarity of the ASN to its closest trellis for each month of the study.

The dimensions and sizes of the long stars within the ASN loosely followed few trends. The stars did have a tendency to grow over time as they obtained more points; however, the higher the number of nodes the star accumulated, the more connected it became until it was too well-connected to be considered a star pattern.

The star ensemble analysis of the network exceeded 50% mark for the first year of the study; however, the number ended up dropping below that mark for the remainder of the study. The star similarity for March 2004 was nearly 65%. The next two months remained close to 60% before a sharp decrease occurs at the end of 2004. Month 15 was the first month in the study in which the star similarity value dropped below 50%. The star similarity percentage reaches its lowest point in March 2009 when it is slightly higher than 34%.

Fig-5a and 5b compare the dominant long star terms from the ASN against the calculated star similarity percentages for each iteration. Both charts show a steady decline in all values over the course of the study. Some differences are notable between the two statistics. The x and x^2 terms of the ASN remain on a decline from period to period; on the other hand, the star similarity percentages display a slight rise in value (less than 2.5%) during several iterations of the network.

E. Trellis Ensemble Similarity Analysis

The visualizations of the ASN showed a separation between the star-like points at the edge and the core of the network. The outer part of the core consists of low degree nodes that appear grid-like. The numbers for degree 3 and degree 4 nodes were fairly high and in proportion similar to the trellis generating function signature. In combination with the low count of degree 2 nodes outside the long stars, it helps support the presence of a pattern from the trellis ensemble within the ASN.

Table II shows the results for the dimensions for the dominant trellis in the network at each iteration, and Table III demonstrates the process used to find the dominant trellises. The p and q values represent height and width of the trellis mesh respectively. The first trellis observed was a 4x5 pattern. Roughly every twelve months, the trellis pattern became slightly larger as the width continually increased by 1. Rather than using a different dominant trellis with a different p value (a 5x6 trellis showed error less than 0.1) from the months before, the experiment was conducted with the 4x10 trellis for the final year of the study. The error in this adjustment did not alter the results much.

Fig-6a and 6b show the relationship between the dominant trellis terms from the ASN and the trellis similarity for each month in the study. The ASN appeared to go through two phases: the first phase in which it increased from 25% to 38% until the midpoint of the study (month 30) and the second phase in which it remained steady between 35% and 38%. The trellis similarity percentages are still below 40%. Side by side the two charts distinctly share a trend. The x^3 term of the ASN and the trellis similarity percentages follow almost exactly the same pattern over the course of the project.

VI. Conclusions

The work makes two main contributions in the area. This paper uses the novel technique of *generating function signature* GFS and extends their usefulness. The main advantages of GFS like techniques are twofold- that they can handle perturbation of target as well as pattern graphs which is very important for analysis of large scale complex networks. The method is also computationally efficient. It can be extended even to mega scale complex networks- if their degree distribution can be sampled with reasonable confidence.

However, GFS do have fundamental limits. Besides the difficulty in pattern separation, generating functions cannot pinpoint the location of a pattern within the topology or determine that the structure exists in one place. Pattern analysis through generating functions is only able to locate signatures of familiar patterns. The nodes belonging to any pattern could be scattered throughout the network without being visible.

This study also reveals another important trend possibly great engineering implications and not reported earlier- that the autonomous systems are becoming more trellis-like and continuously moving away from this single-home or star-like configuration. This transformation has been rapid from the very beginning of this study to the middle of 2006. Finally, there seems to be a fundamental transformation of the relative weights of edge and core ASN network where the ratio has shifted from roughly 70:30 to about 45:55 based on star and trellis similarity.

What are the implications of such trend? This pattern of evolution of ASN indeed call for a new regime of routing and transport protocols- than currently one- which can take advantage of increased parallel paths and regularized topology based communication. This indicates it might be time for a suit of next generation protocols (across layers) in the line of parallel communications in Internetworking.

References

1. "Autonomous System Number Allocation", http://lacnic.net/documentos/lacniciii/chapter-6-en.pdf
2. A. Vazquez, R. Pastor-Satorras, A. Vespignani, "Internet Topology at the Router and Autonomous System Level", arXiv:cond-mat/0206084v1, 2002.
3. S.-H. Yook, H. Jeong, A.-L. Barabási, "Modeling the Internet's Large Scale Topology", Proceedings of the National Academy of Sciences of the United States of America, 2002.
4. D. Magoni, J.J. Pansiot, "Analysis of the Autonomous System Network Topology", ACM SIGCOMM Computer Communication Review, 2001 pp. 26-37.
5. B. Zhang, R. Liu, D. Massey, L. Zhang, "Collecting the Internet AS-level Topology", ACM SIGCOMM Computer Communication Review, Vol. 35, Issue 1, 2005, pp. 53-61.
6. A. Dhamdhere, C. Dovrolis, "Ten Years in the Evolution of the Internet Ecosystem", Proceedings of the 8th ACM SIGCOMM conference on Internet measurement, Vouliagmeni, Greece, 2008. pp. 183-196.
7. R. G. Clegg, R. Landa, M. Rio, U. Harder. "A Likelihood Based Framework for Assessing Network Evolution Models Tested on Real Network Data", Proceedings of the 1st Annual Workshop on Simplifying Complex Network For Practitioners, Venice, Italy, July 2009. SIMPLEX '09, ACM, New York, NY, 1-6.
8. R. Oliveira, B. Zhang, L. Zhang, "Observing the Evolution of Internet AS Topology", ACM SIGCOMM, Kyoto, Japan, August 2007.
9. D. S. Lee, J. L. Kalb, "Network Topology Analysis", Sandia National Laboratories, 2008.
10. J. Wang, P. Wilde, H. Wang, "Topological Analysis of a Two Coupled Evolving Networks Model for Business Systems", Expert Systems with Applications, Vol. 36, Issue 5, 2009, pp. 9548-9556.
11. M. Chatterjee, S. K. Das, D. Turgut, "WCA: A Weighted Clustering Algorithm for Mobile Ad Hoc Networks", Cluster Computing, Vol. 5, Issue 2, 2002, pp. 193-204.
12. C. Xie, G. Chen, A. Vandenburg, Y. Pan, "Analysis of Hybrid P2P Overlay Network Topology", Computer Communications, Vol. 31, Issue 2, 2008, pp. 190-200.
13. R. Oliveira, J. Park, B. Zhang, "Internet Topology Collection", http://irl.cs.ucla.edu/topology/.
14. C. Stewart & J. I. Khan, A Large Scale Evolutionary Analysis of the Internet Autonomous System Network, Proceedings of the 9th International Conference on Information Technology- New Generations, ITNG 2012, Las Vegas, Nevada, USA , April 16-18, 2012.
15. M. E. J. Newman, S. H. Strogatz, D. J. Watts, "Random Graphs with Arbitary Degree Distribution and Their Applications", American Physical Society Review E 64.026118, 2001.
16. D. S. Callaway, M. E. J. Newman, S. H. Strogatz, D. J. Watts, "Network Robustness and Fragility: Percolation on Random Graphs", American Physical Society Review Letter 85, 2000.
17. S. N. Dorogovtsev, J. F. F. Mendes, "Evolution of Networks", Advances in Physics, 2001.
18. R. Oliveira, D, Pei, W. Willinger, B. Zhang, L. Zhang, "Quantifying the Completeness of the Observed Internet AS-level Structure", UCLA Computer Science Department.

SESSION

ULTRA LOW POWER DATA-DRIVEN NETWORKING SYSTEM, ULP-DDNS, ARCHITECTURE AND IMPLEMENTATION; CURRENT STATUS AND FUTURE DIRECTION

Chair(s)

Prof. Hiroaki Nishikawa

A Comprehensive Evaluation of ULP-DDNS by Platform Simulator

Kazuhiro Aoki[1], **Hiroshi Ishii**[2], **Makoto Iwata**[3] **and Hiroaki Nishikawa**[4]
[1]Information Infrastructure Laboratory, Inc., Tsukuba Science City, Ibaraki, JAPAN
[2]School of Information and Telecommunication Engineering, Tokai University, Minato, Tokyo, JAPAN
[3]School of Information, Kochi University of Technology, Kami, Kochi, JAPAN
[4]Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba Science City, Ibaraki, JAPAN

Abstract— *It is essential to keep networking system available in emergency. Power consumption will therefore be one of the most important issues to realize both platform and communication environment. This paper describes about comprehensive evaluation of ultra-low-power data-driven networking system (ULP-DDNS). ULP-DDNS, which is a research project, is aiming at development of data-driven networking system which can achieve ultra-low-power consumption: 1/300 less than the present system. This paper firstly reports the effect of power saving schemes in ULP-DDNS. The authors have implemented ULP-DDNS node and platform simulator for evaluation of ULP-DDNS comprehensively. This paper then describes evaluation scheme of ULP-DDNS with ULP-DDNS node and simulators. Furthermore, this paper describes an implementation of ULP-DDCMP platform simulator and shows current status of experimental study. Finally, the authors propose data-driven load balancing scheme to maintain the networking system in working without over-loaded state.*

Keywords: ultra-low-power, data-driven principle, networking architecture, self-timed elastic pipeline, chip multiprocessor

1. Introduction

For rapid evacuation, it is important to keep communication environment available in emergency. It is necessary to realize such robust environment because some earthquakes and tornadoes occur in Japan in a few years. Infrastructureless communication environment can be then supposed in emergency. And power consumption is crucial issue to keep communication environment available for a long time as possible [1]-[3].

Currently, so-called pervasive networking environment as social infrastructure has been widely studied [4]. Furthermore, there are many studies about mobile ad hoc network[5] which is an infrastructureless network and is a group of wireless devices that organize themselves in a mesh topology to find routes and relay packets from the hardware platform through the network layer to application. Considering the case where next generation of pervasive networking is realized over ad hoc network suitable for emergency and some tentative accidents, some of authors study data-driven implementation of ad hoc communication environment [6].

The authors have started research project named "ultra-low-power data-driven networking system (ULP-DDNS)"[7]. ULP-DDNS project is aiming at development of data-driven networking system which can achieve ultra-low-power consumption: 1/300 less than the present system.

This paper firstly reports the effect of power saving schemes in ULP-DDNS. In ad hoc networking application layer, we have evaluated reducing the number of packets in broadcasting with load-aware flooding scheme. On platform, the authors have studied effective protocol handling such as data-driven implementation of UDP/IP, and optimized circular pipeline. Furthermore, this paper reports power saving scheme such as power gating and runtime voltage scaling on the platform. Toward comprehensive evaluation of ULP-DDNS, the authors have implemented ULP-DDNS node and platform simulator. This paper then describes evaluation scheme of ULP-DDNS. In ad hoc networking application layer, the authors have evaluated traffic in all of the ad hoc network with network simulator. We use logs of the network simulator as input of platform simulator for comprehensive evaluation. Furthermore, this paper describes an implementation of ULP-DDCMP platform simulator and shows current status of experimental study. Finally, the authors discuss about data-driven load balancing scheme to maintain the networking system in working without over-loaded state.

2. Power Saving Schemes in ULP-DDNS

This section reports power saving schemes in each layer of ULP-DDNS. Fig. 1 shows layer of ULP-DDNS. A node of ULP-DDNS is a platform which is used data-driven

chip multiprocessor(UDP-DDCMP) and self-timed elastic pipeline(ULP-STP). Runtime voltage scaling is implemented in the platform for ultra low power consumption. Ad hoc networking Application and UDP/IP handling is also implemented on the platform.

The method of reducing the number of UDP packet in ad hoc networking application layer and reducing power consumption on platform is described in following sections.

2.1 Power Saving Schemes of Ad hoc Networking Architecture

The authors have studied mobile ad hoc network as an applicable network architecture to disaster situation. In disaster situation, effective information discovery is firstly important. At a same time, effective secure communication is needed. They should be realized under the effective data transfer on ad hoc network. We have proposed these schemes in ultra low power consumption. Refer to [8] about these schemes. This paper reports about load-aware flooding[9] because this scheme is used in all communication on ad hoc network.

When an ad hoc network is to be used in a disaster situation, it is likely that emergency information will be broadcast by voice streaming from a small number of nodes. Conventionally, simple flooding (SF) has been used as a method of broadcasting streams to the entire network. However, if SF is applied to information flows in which packets are generated at a high rate, as is the case with voice streams, packet loss will occur frequently, causing degradation in the quality of service.

As a method of broadcast streaming, we have already proposed a Load-aware Dynamic Counter-based Flooding (LDCF), and showed that LDCF results in fewer unnecessary packets being sent and less degradation in quality of service (QoS) than SF or other methods. Furthermore we showed that the application of LDCF to broadcast voice streaming in an ad hoc network results in fewer instances of packet loss than SF. Nodes used in an ad hoc network are normally powered by a battery of finite capacity. Therefore a reduction of the number of exchanged packets is desirable, not only because of the reduction in traffic load itself, but also because it results in reduced power consumption. As a results, we have evaluated that LDCF generates less traffic and consumes less power per node than SF.

2.2 Data-Driven Implementation of UDP/IP for ULP-DDCMP

The authors have studied the effectiveness of an implementation of protocol offloader using networking-oriented

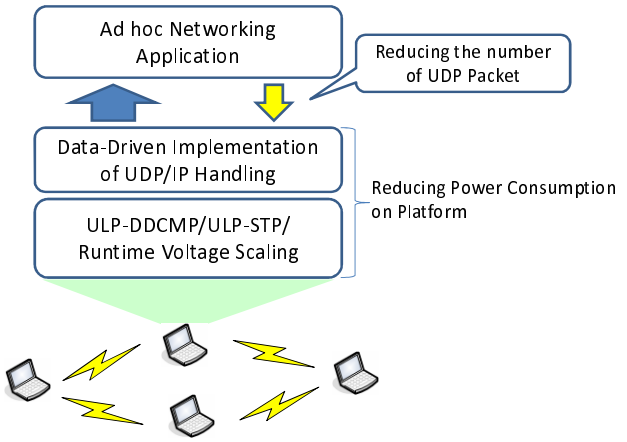


Fig. 1: Power Saving Schemes on Platform

data-driven processors CUE (Coordinating User's requirements and Engineering constraints) designed by CUE project [10], [11]. In ad hoc network, it is necessary to realize connection-less protocol such as UDP/IP for flexible routing and realtime communication. We proposed data-driven implementation of UDP/IP for ULP-DDCMP to minimize overheads in protocol handling.

Then we studied architecture of CUE-v2/CUE-v3 for ad hoc networking environment [12], [13]. And we evaluated effectiveness of protocol handling offloader using CUE processor system. It shows that data-driven protocol offloader can keep minimum turn-around time in comparison with conventional PC(Personal Computer) [14]. We realize header processing and data processing concurrently utilizing multiprocessing capability of data-driven processor without runtime overheads.

This paper shows evaluation of UDP/IP on ULP-DDCMP in power consumption in Section 3.2

2.3 Data-Driven Chip Multiprocessor

2.3.1 Self-Timed Power-Aware Elastic Pipeline:ULP-STP

The authors have studied data-driven chip multiprocessor based on self-timed elastic pipeline for ultra-low-power in realtime multiprocessing.

In data-driven chip multiprocessor, both the dynamic and static power dissipations are minimized by distributing the processing load over multiple processing cores which is slowed down by using runtime/dynamic voltage scaling (DVS) technique as long as the required processing speed is satisfied[15]. In addition, the voltage-supply to idle circuit blocks or cores is cut by using fine-grained power gating

(PG) technique[16]. It is therefore intended that the ultra-low-power DDCMP would be implemented by self-timed power-aware elastic pipeline named ultra-low-power self-timed pipeline (ULP-STP)[17]. Because of self-timed elastic data-transfer mechanism of the original STP, it can work well under variable voltage without adjusting clock frequency even if the altered voltage could transiently fluctuate at individual pipeline stage. Since the pipeline throughput can be adaptive to its processing load only by altering supply-voltage appropriately, a power-aware pipeline scheme can be realized naturally in terms of dynamic power saving. For instance, proportional-integral differential (PID) control method can be applied to such voltage control by monitoring consumption current of a target power domain within the chip. The STP is also suitable for gating power-supply to fine grain circuits since its stage-by-stage data-transfer control independently activates only pipeline stages with valid data. We therefore proposed a stage-by-stage power gating scheme adopted in the STP. This scheme provides natural signal gating, i.e., it stops the unnecessary signal propagation and transistor-switching at pipeline stage level without any global control mechanisms resulting in both power dissipation and processing speed degradation. Moreover, it makes it possible to scale the voltage even when the stages are activated because it can be realized without any global oscillator such as phase-locked loop (PLL) circuit, which forces pipeline flush ahead of the frequency and voltage change. In order to analyze the low-power characteristics of the ULP-STP and to estimate power-performance of various ULP-STP based systems, an experimental LSI chip has been fabricated by using 65 nm CMOS process. We have implemented ultra-low-power data-driven chip multi processor: ULP-DDCMP based on evaluation of the experimental LSI.

2.3.2 An Implementation of Ultra-Low-Power Data-Driven Chip Multiprocessor: ULP-DDCMP

In our previous study on the data-driven processor, it is already revealed that the networking protocol handling consists of several sequential processing parts such as a processing part to generate output data by reading an array sequentially. To prevent such sequential processing part from being the bottleneck of the networking protocol handling, some of the authors has already proposed a hybrid processor architecture in which a control-driven instruction execution is realized by out-of-order execution scheme suitable for the sequential processing in addition to the ordinary data-driven instruction execution[13]. Based on this previous study, it is revealed that the sequential processing parts should be efficiently executed in parallel with the execution of the other non-sequential processing parts. However, from a viewpoint of low-power processing, the previously proposed hybrid architecture is unsuited for the ULP-DDNS, because the control-driven execution requires additional controls to execute instructions in contrast to the pure data-driven processor. Therefore, the ULP-DDCMP is designed to exploit the data-driven principle and its instruction execution pipeline is discussed in this paper by focusing on the utilization frequency of the instruction execution pipeline stages activated by unary-operation instructions which are the main constituent of the sequential processing parts. In the ULP-DDCMP, a processor core is realized by a circular pipeline necessary for instruction execution, and each pipeline stage is activated or driven only when a packetized data is transferred, and thus signal gating at pipeline stage level is naturally realized without any additional circuit[18]. In fact, ULP-DDCMP has no dynamic power dissipation. On the other hand, it is important to reduce the amount of leakage current which is the dominant cause of the increase of static power dissipation, in order to exploit the benefits of the scaling of transistor to the improvement of the performance of the ULP-DDCMP. We focused on the utilization frequency of the instruction execution modules, and have proposed an instruction execution pipeline structure in which infrequently used modules are bypassed and/or powered off to shorten the instruction execution time and to reduce static power dissipation. The effectiveness of the proposed pipeline structure is estimated based on a prototype LSI implementation.

The authors have studied reasonable mechanism for avoid over-loaded state by using processor core configuration of ULP-DDCMP and observability of ULP-STP. Conventional schemes for avoid over-loaded state is relatively much power consumption in each service because restriction is excess for performance. We proposed mechanism which is combined DVS function and I/O control for observation of load with load-distribution by round-robin. It is effective to realize ultra-low-power networking environment which is suitable for demanded performance. In addition, we have developed ULP-DDNS node. Refer also [18] about this node in detail.

3. Comprehensive Evaluation of Power Saving Scheme by Platform Simulator

This section proposes comprehensive evaluation scheme with network simulator and platform simulator. Fig. 2 shows image of summation of power consumption. Network simulator have already used evaluation in ad hoc networking application. As comprehensive evaluation, we proposes using logs as a network simulation result to input of the

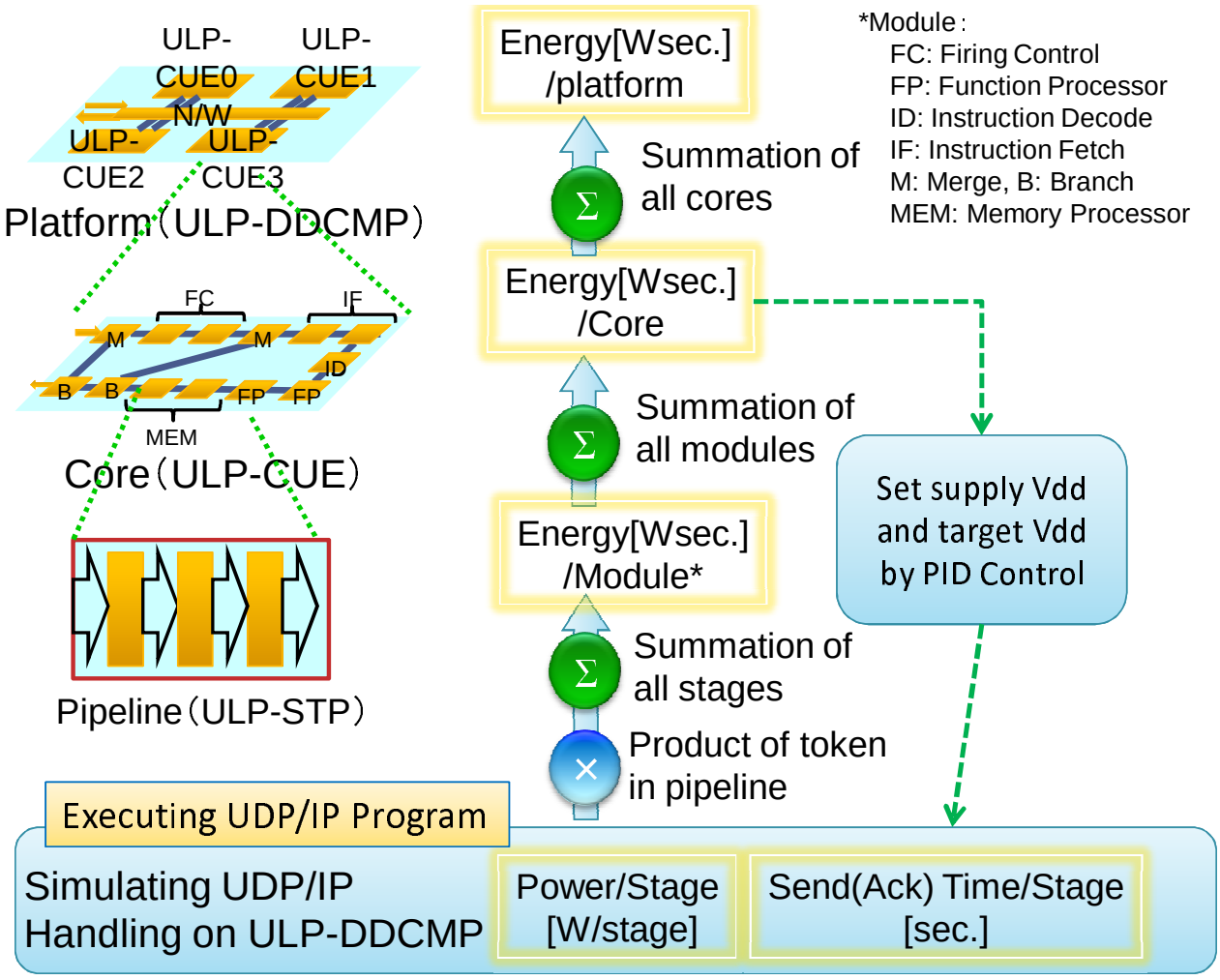


Fig. 2: Comprehensive Evaluation of ULP-DDNS

platform simulator. Platform simulator can evaluate power consumption and turn-around time of program on data-driven chip multiprocessor. Platform simulator is necessary to evaluate energy of platform in UDP/IP handling because analog electronic circuit simulator which is SPICE is too complicated to measure power and turn-around time in UDP/IP handling according to network simulation log. In execution program, logs which has input time and data length can be used as input for the platform. Summation power consumption in each platform is a total power consumption of ULP-DDNS.

Platform simulator has hierarchy among self-timed elastic pipeline (ULP-STP), cores (ULP-CUE), and platform (ULP-DDCMP) as shown in Fig. 2. Firstly, platform simulator evaluates power consumption and switching time of stages of ULP-STP. Power in each stage and send/ack time between stages are then derived from ULP-DDNS node, prototype of ULP-STP and gate simulation. In PID control, time in change supply voltage and target Vdd is simulated by platform simulator. Parameters such as some gain is tuned by data of gate simulation. Platform simulator sums up energy of stages of which a module consist. ULP-CUE consists of some module such as firing control (FC), function processor (FP), instruction decode (ID), etc. All modules of which ULP-CUE consists is connected as a circular pipeline. Platform simulator also sums up energy of all stages of a circular pipeline which is elements of an ULP-CUE. Furthermore, the simulator sums up energy of all cores on an ULP-DDCMP. It is power consumption of a platform.

3.1 Platform Simulator for ULP Platform

It is necessary to validate power consumption of ULP-DDNS in detail. So the authors proposed power simulator/validator as a platform simulator[19]. Then, we have

implemented the platform simulator. This paper shows how to use of the platform simulator.

Fig. 3 shows input to platform simulator and output from it. Platform simulator requires schedule of token stream as an input. This schedule is generated from network simulation logs. Platform simulator also demands UDP/IP program to evaluate power consumption in UDP/IP handling. Platform simulator has a topology of platform, a topology of core and parameters which indicate specification in each stage such as power and switching time in order to calculate power consumption of platform. Platform simulator then outputs energy and turn-around time in UDP/IP Handling.

Fig. 4 shows pipeline structure in the simulator and simulation results. Fig. 4(a) shows a circular pipeline structure of ULP-CUE. The platform simulator can freely design pipeline structure of core and router in a platform. Then, the simulator can set parameters power and time which is evaluated in a platform flexibly.

When input datagrams concretely create, it is necessary to collaborate with network simulator which evaluates networking architecture. The authors have studied how to communicate between our simulator and network simulator. We uses data log in network simulator can use information of data input in our simulator. On the other hand, specification of CMP in our platform simulator may apply to a node in network simulator.

Multi pipeline loop of ULP-CUE is applied to pipeline structure of core in this simulation because ULP-CUE has same pipeline. Fig. 4(b) shows graph as a simulation result. X axis of the graph is execution time of UDP/IP on a platform which is target of simulation. And y axis of the graph is power consumption in each time. We evaluate and tune a platform on ULP-DDNS, and show effectiveness in ultra-low-power of ULP-DDNS. Then, it is necessary to adjust parameters in PID control to runtime/dynamic voltage scaling mechanism. So the authors have been simulating UDP/IP on ULP-DDCMP architecture in several settings of PID control by using platform simulator.

3.2 Evaluation of ULP Platform

This section reports current status of the comprehensive evaluation. The authors have validated logs of network simulation. Fig. 5 shows percentage of sending packets in LDCF to SF. X axis of the graph is node ID which identify node on the network. Y axis of the graph is percentage of sending packet in LDCF to SF. Almost node is less than 10% of SF in sending packet. Fig. 6 shows the distribution of its percentage. X axis of graph is percentage of sending packet in LDCF to SF. And y axis of the graph is the number

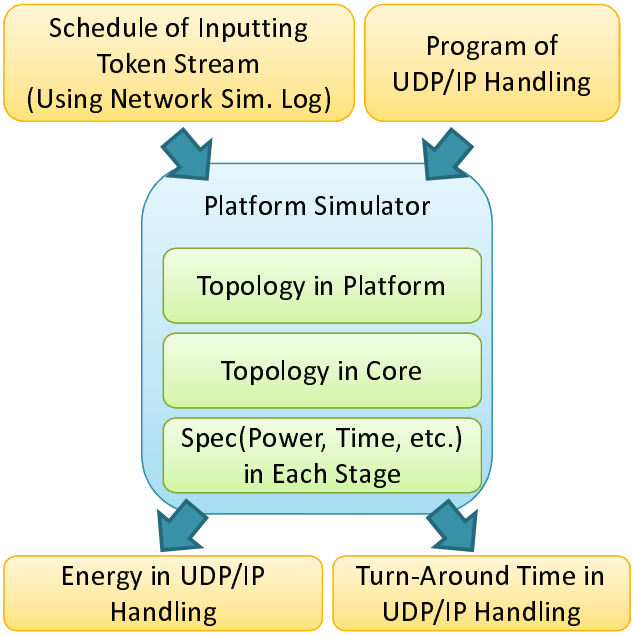
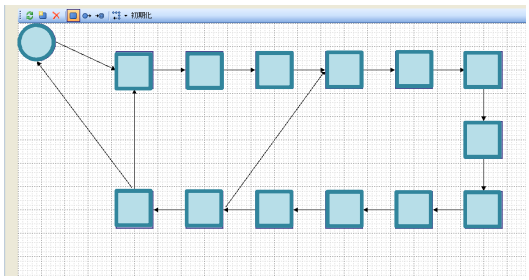


Fig. 3: Specification of Platform Simulator

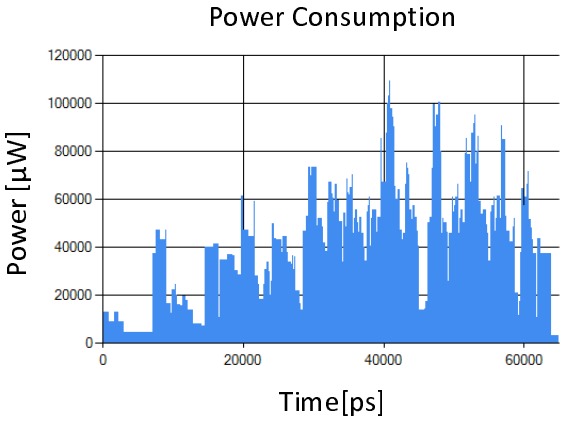
of nodes of which sending packet is in each percentage. For comprehensive evaluation, 10 nodes is enough to sampling node in logs of network simulator because of estimation in confidential interval.

Furthermore, the authors have evaluated power consumption of conventional processor to compare power consumption with ULP-DDCMP. ULP-DDNS node has Atom as a application processor which driven Linux OS. So, we have evaluated power consumption of Atom on ULP-DDNS node. On Atom, it is necessary to drive OS in order to handle UDP/IP. To evaluate power consumption in just UDP/IP handling, we firstly measured power consumption in stand by mode of Atom. Its power was 210.7 μWsec./packet. We then measured power consumption in UDP/IP handling on Atom. Its power was 232.6μWsec./packet. Thus, power in just UDP/IP handling is 21.9 μWsec./packet. Then we need to consider power consumption of L2 cache because L2 cache is not used in UDP/IP handling. We estimated power consumption of L2 cache and deduct it from measured power consumption. As a result, power consumption of L2 cache is 10.9 μWsec. because ratio of circuit area of L2 cache to a chip is less than 1/2. Therefore, we have evaluated that power in UDP/IP handling on Atom is 11.0 μWsec./packet.

On the other hand, power in UDP/IP handling on ULP-DDCMP is 3.2 μWsec./packet. Hence, power consumption of ULP-DDCMP is 1/3 of Atom in UDP/IP handling. So we evaluated that ULP-DDCMP is enough to have ultra-



(a) Configuration of Circular Pipeline



(b) Evaluation Result

Fig. 4: Power of UDP/IP on ULP-DDCMP

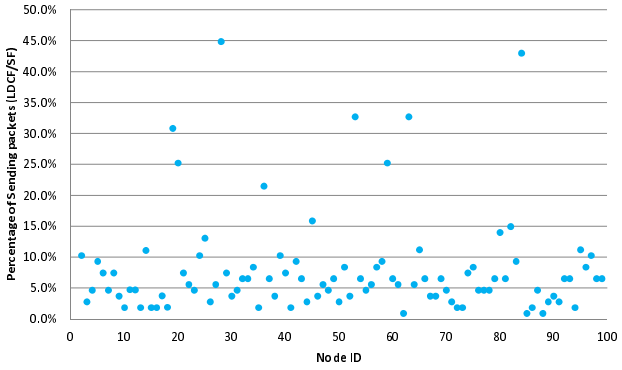


Fig. 5: Percentage of Sending Packets in LDCF to SF

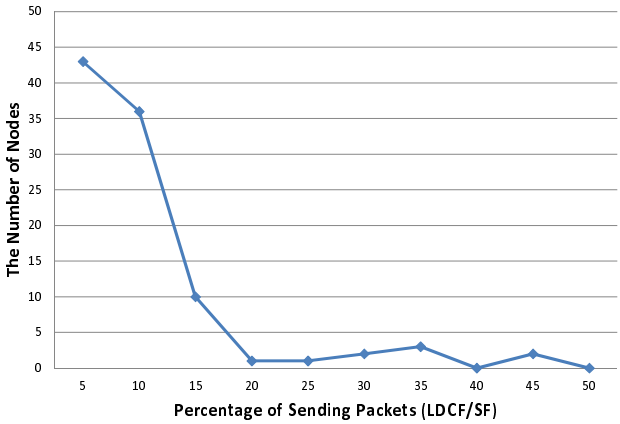


Fig. 6: Distribution in Percentage of Sending Packets in LDCF to SF

low-power effect.

As a future direction, we will firstly analyze UDP/IP handling on ULP-DDCMP when packets input according to time of network simulation log. This simulation result can be compare with power consumption on ULP-DDNS node which handle UDP/IP on ULP-DDCMP. At the same time, we will tune simulation model of power consumption more concretely. For example, it is necessary to tune power consumption model in DVS because current model in platform simulator is too simple to evaluate power consumption in DVS. Furthermore, tuning instruction set on ULP-DDCMP is important to save power. We then need to study offloading scheme, which is to optimize executing application on ULP-DDCMP. At least, it is necessary to implement ad hoc networking application on ULP-DDCMP because we verificate evaluation of platform simulator whether it match the evaluation of network simulator. And the authors have studied implementation of overload avoidance function on ULP-DDCMP. Refer to [20], [21] about these study.

4. Conclusion

This paper firstly reports power saving schemes of each layer in ULP-DDNS project. It is clear to achieve high effect in reducing power consumption with proposed schemes. The authors then proposed comprehensive evaluation method which uses output of network simulator as input of platform simulator. We showed that power consumption of ULP-DDNS can be evaluated by its cooperation. Furthermore, we analyzed logs of network simulator, and we showed

the distribution of percentage of sending packet in LDCF to SF. As a standpoint of estimation considering confidential interval, 10 nodes is enough as sample nodes for the evaluation. This paper describes implementation of platform simulator, and reports current status and future direction of the comprehensive evaluation.

As a further works, the authors will comprehensively evaluate power consumption of ULP-DDCMP as shown in previous section. And Tuning platform simulator will be needed to evaluate power consumption precisely. Furthermore, we have studied overload avoidance function in platform and networking system to realize robust networking system.

Acknowledgments

Although it is impossible to give credit individually to all those who organized and supported the CUE project and the ULP-DDNS project, the authors would like to express their sincere appreciation to all the colleagues in the project.

The CUE project and the ULP-DDNS project are partially supported by Core Research for Evolutional Science and Technology (CREST), Japan Science and Technology Agency, Strategic Information and Communications R&D Promotion Programme (SCOPE), Ministry of Internal Affairs and Communications, Japan, the Grants-in-Aid for Scientific Research of Japan Society for the Promotion of Science and Semiconductor Technology Academic Research Center (STARC). And, this work is supported by VLSI Design and Education Center(VDEC), the University of Tokyo in collaboration with Synopsys, Inc. and Cadence Design Systems, Inc.

References

[1] T. Inagaki and S. Ishihara, "HGAF: A Power Saving Scheme for Wireless Sensor Networks," IPSJ Journal Vol.50, No.10, pp. 2520–2531, Oct. 2009.

[2] A. Keshavarz-Haddad and R. Riedi, "Bounds on the benefit of network coding: Throughput and energy saving in wireless networks," IEEE INFOCOM 2008, Phoenix, Arizona, USA, pp. 376–384, April 2008.

[3] L. Sukyoung, K. Laeyoung and K. Hojin, "MIPv6-Based Power Saving Scheme in Integrated WLAN and Cellular Networks," IEICE transactions on communications Vol. E90-B, No. 10, pp. 2780–2783, Oct. 2007.

[4] Debashis Saha and Amitava Mukherjee, “Pervasive Computing:A Paradigm for the 21st Century,”IEEE Computer, Vol. 36, No. 3, pp. 25–31, Mar. 2003.

[5] Jie Wu, Ivan Stojmenovic, “Ad Hoc Networks,” IEEE Computer, Vol.37, No.2, pp.29–31, Feb. 2004.

[6] Hiroshi Ishii, Chee Onn Chow, Masahiro Yamamoto, Hiroaki Nishikawa, “Ad hoc and Ubiquitous Communication Environment supported by Data-Driven Networking Processor,” IEEE TENCON 2006, Hong Kong, China, Nov. 2006

[7] Hiroaki Nishikawa, Hiroshi Ishii, and Makoto Iwata, “Collaborative Research Project on Ultra-Low-Power Data-Driven Networking System,” Proc. of the 2008 Int’l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 697–703, July 2008.

[8] Hiroshi Ishii, Keisuke Utsu and Hiroaki Nishikawa “Integrated Evaluation on Effectiveness of ULP-DDNS Networking Layer,” Proc. of the 2012 Int’l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP6026, July 2012.

[9] Keisuke Utsu, Hiroaki Nishikawa, and Hiroshi Ishii, “Performance Evaluation of Load and Battery Charge Oriented Broadcast Streaming Method over Ad Hoc Networks,” Proc. of the 2012 Int’l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP6027, July 2012.

[10] Hiroaki Nishikawa, “Design Philosophy of a Networking-Oriented Data-Driven Processor-CUE,” IEICE Trans. Electron., vol.E89-C,no.3, pp.221–229, Mar. 2006

[11] Hiroaki Nishikawa, Hiroshi Ishii, Makoto Iwata, and Kazuhiro Aoki, "An Offloading Scheme for Ultra Low Power Data-Driven Networking System", Proc. of the 2009 Int’l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 595–601, July 2009.

[12] Shinya Ito, Shouhei Nomoto, Hiroshi Tomiyasu and Hiroaki Nishikawa, “The Microarchitecture of the CUE-v2 Processor: Enabling the Simultaneous Processing of Dataflow and Control-Flow Threads,” Proc. 2004 Int’l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 525–531, June 2004.

[13] Hiroaki Nishikawa, Hiroshi Tomiyasu, Masanobu Okamoto, Masayoshi Sugiyama, Hiroyuki Uchida, Osamu Mizuno, Hiroshi Ishii, Makoto Iwata, “CUE-v3: Data-Driven Chip Multi-Processor for Ad hoc and Ubiquitous Networking Environment,” Proc. of the 2007 Int’l Conf. on Parallel and Distributed Processing Techniques and Applications, pp.623–629, June 2007.

[14] Kazuhiro Aoki, Hiroshi Ishii, Osamu Mizuno, Makoto Iwata and Hiroaki Nishikawa, “Data-Driven Protocol Off-Loading for Ad Hoc Networking Environment,” Proc. of the 2008 Int’l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 662–668, July 2008.

[15] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen, “Low Power CMOS Digital Design,” IEEE Trans. on Solid-state Circuits., vol. 27, No. 4, pp.473–483, Apr. 1992.

[16] Shin-ichiro Mutoh, Satoshi Shigematsu, Yoshinori Gotoh, Shinsuke Konaka, “Design Method of MTCMOS Power Switch for Low-Voltage High-Speed LSIs,” Proc. of Asia and South Pacific Design Automation Conference, Hong Kong, pp.113–116, Jan. 1999.

[17] Kei Miyagi, Shuji Sannomiya, Makoto Iwata and Hiroaki Nishikawa, “Low-Powered Self-Timed Pipeline with Runtime Fine-Grain Power Supply,” Proc. of the 2012 Int’l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP6029, July 2012.

[18] Shuji Sannomiya, Kazuhiro Aoki, Makoto Iwata and Hiroaki Nishikawa, “Power-Performance Verification of Ultra-Low-Power Data-Driven Networking Processor: ULP-CUE,” Proc. of the 2012 Int’l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP6028, July 2012.

[19] Hiroaki Nishikawa, Kazuhiro Aoki, Hiroshi Ishii, and Makoto Iwata, “Intermediate Achievement of Ultra-Low-Power Data-Driven Networking System: ULP-DDNS,” Proc. of the 2011 Int’l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 421–427, July 2011.

[20] Yukikuni Nishida and Hiroaki Nishikawa, “A Study on Overload-Avoidance Scheme of ULP-DDNS for Congestion-Free Networking System,” Proc. of the 2012 Int’l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP6030, July 2012.

[21] Hideki Yamauchi and Hiroaki Nishikawa, “Proposal of Applying ULP-DDNS to Congestion-Free Networking System,” Proc. of the 2012 Int’l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP6031, July 2012.

Integrated Evaluation on Effectiveness of ULP-DDNS Networking Layer

Hiroshi Ishii[1], **Keisuke Utsu**[1], **and Hiroaki Nishikawa**[2]
[1]Department of Communication and Network Engineering,
School of Information and Telecommunication Engineering, Tokai University, Minato, Tokyo, Japan
[2]Department of Computer Science, Graduate School of Systems and Information Engineering,
University of Tsukuba, Tsukuba, Ibaraki, Japan

Abstract – *This paper describes the up-to-date results got through Ultra Low-Power Data-Driven Networking System (ULPDDNS) project mainly about the network portion. The ULPDDNS has been aiming at achieving reduction of power consumption to 1/100 – 1/1000 compared with the existing systems. We have been tackling with the networks applicable to the emergency situation such as natural disaster. In the emergency case, the need of low power consumption systems and networks are really essential due to lack of power infrastructure. We believe the most applicable network to disaster situation is Mobile Ad hoc NETwork (MANET). This paper shows the evaluation results on the ULPDDNS networking layer activities on MANET by integrating three techniques to establish highly efficient and energy saving MANETs with reasonable performance: (1) GPS-aided target information discovery, (2) Load-aware broadcast-type contents delivery, and (3) Trust relationship list based key management.*

Keywords: Mobile Ad hoc Network, broadcasting, information discovery, trust management

1 Introduction

The ULPDDNS consists of mainly two parts: one is the platform (processor) and the other is the network. Our objective is to develop a data-driven networking system that can achieve reduction of power consumption to 1/100-1/1000 compared with the existing systems, especially in the situation just after a disaster happens.

We believe the most applicable network architecture to disaster situation is Mobile Ad hoc NETwork (MANET) [1][2]. A MANET is defined as a group of wireless devices that are capable of organizing themselves in a mesh topology in order to find routes and relay packets from each node to any other node within the network without a support of a pre-installed infrastructure. The deployment of these networks is expected to take place in critical environments such as during disaster or on battlefield, where pre-installed infrastructure is not available. We assume the network portion of ULPDDNS is to be MANET and have been studying MANET architecture that can achieve the high efficiency to reduce power consumption with keeping possibly high performance. In order to make both network efficiency and performance (QoS and security) better, we have to consider that emergency network has two network usage phases such as (1) network setting-up phase (just after a disaster happened) and (2) stable phase.

Regarding the intermediate reports of our activities has already been shown in [3]. There we had three main approaches, (1) GPS-aided target information discovery, (2) Load-aware broadcast-type contents delivery, and (3) Trust relationship list based key management. This paper will show the integrated effects of those three approaches as the final stage evaluation of our ULPDDNS project. It firstly shows three main approaches we have been proposing one by one. Then it shows the each result got so far for each approach. Then, assuming a traffic model that could be realistic in case of emergency, the paper shows the integrated evaluation results of our approaches.

2 MANET for disaster

Although we have already reported in [3] the relation of MANET to disaster situation and possible solutions, we would like to summarize them here.

2.1 Applicability of MANET to the disaster situation

As shown above, MANET has no infrastructure, no centralized controllability, dynamically changing network topology, and multi-hop information transmission. It is an infrastructure-less and a group of wireless and mobile devices that organize themselves, can be an alternative to the existing network infrastructure in case of emergency and/or events, and will play an important role for achieving low power consumption network. Our target is reduction of "number of packets (redundant packet transmission)" to 1/10 of the present in case of emergency (just after disaster happens). MANET is most applicable to the disaster situation and the situation needs ultra low power consumption to make battery have longer life time.

After the disaster, we have to consider network life time. We consider following two phases.

- Phase 1 (transient): This phase means that of just after the disaster happened. Several nodes in some area may organize a MANET but each node may not know IDs of other nodes such as IP addresses. So, usual routing (unicast or multicast data transfer) cannot be executed. In this phase, however, useful information must be sent all over the network urgently such as "where is a refuge, hospital?", "how is my house?", and so on. Both "Pull" (Information discovering) and "Push" (Information broadcasting) are needed without use of IP addresses.

- Phase 2 (stable): In this phase, each node has each routing table and the network becomes stable. Usual communication based on routing with possibly high QoS and security will be required. We have clarified main three study issues in association with two phases above after MANET setting up.

As for phase 1, "effective information discovery" without IP addresses is needed because people in the network, just after the disaster, would probably need such information as refuges, hospitals, and his own house. For this purpose, we have proposed "a GPS based discovery mechanism" [4][5]with reduced control packets. For phase 1, "Effective data transfer" is also needed because people who have some useful information would like to let as many people as possible know the information without IP addresses. For this purpose, we have proposed "Load-aware flooding". This flooding mechanism needs no IP address and can reduce redundant packets. Of course the flooding can be used in phase 2 also.

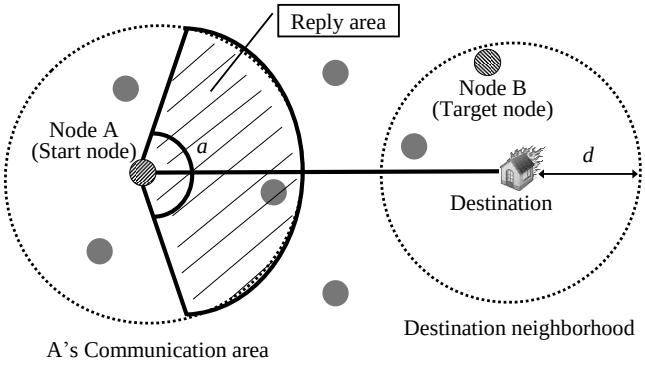
As for phase 2, networks become stable and usual communication by use of IP addresses will be executed. In some cases, "effective secure communication" with ciphering will be needed. For this purpose, we have proposed "Self-organized key management based on trust relationship list".

Below, we show how these mechanisms are effective.

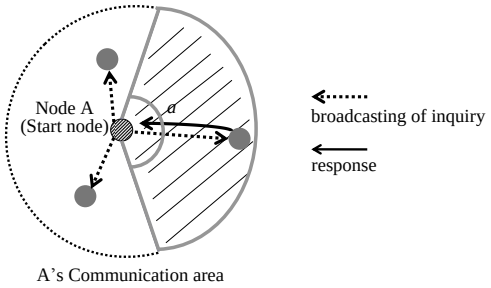
2.2 GPS-aided target information discovery

In case of an emergency phase 1 (just after the disaster happens), local information is probably needed. For instance, for people in the MANET, information about refuges, hospitals and their own houses is urgently needed. This fact requires some information or contents discovery mechanism. Those who need such information would like to inquire other people in the MANET about the information. However, in phase 1, there are some constraints, one of which is that each node address is not well known to other nodes and usual routing based communication (unicasting and multicasting) is unavailable to use. In this situation, only available way of inquiry is using "Simple Flooding (SF)." The SF mechanism is as follows: Simple Flooding is used to deliver messages to the whole network. A node that originates a packet broadcasts it. The packet reaches all the nodes that exist within the area covered by the radio wave transmitted by the origin node. A node that has received the packet re-broadcasts it. This process is repeated by subsequent nodes until the packet reaches all the nodes in the network. In the information discovery, a node sends an inquiry packet to the whole nodes using SF. But, multiplication and contention of inquiry packets will occur and result in waste of electric power. Hence, Effective information discovery with reducing wasteful inquiry packets is needed.

We have proposed a discovery method, which uses location coordinates of nodes using Global Positioning System (GPS), and successive queries and replies between adjacent nodes [4][5]. This method assumes that (i) a node inquiring information (start node) knows the location of destination of which the starting node needs information, for example by use of a map, (ii) some node in the neighbor area of the destination may have the information about the destination which itself is not a ad hoc node, and (iii) each node has GPS terminal. Fig.1 shows a rough sketch of our proposed procedure. To prevent the widespread diffusion of messages in the network which is the major problem with SF, the proposed method uses the location information so that reply packets are sent in response to queries only which have been sent in the pre-defined reply area (hatched area in Fig. 1) direction of the destination.



(a) Outline of the proposed method (Step 1)



(b) Outline of the proposed method (Step 2)

Fig.1 Outline of the GPS based discovery mechanism

By the computer simulation, it is clarified that this method reduces to 25% of simple flooding in best case both the number of packets transmitted/received and the power

consumed in the entire network (Fig. 2), and showed power needed to GPS operation is negligible.

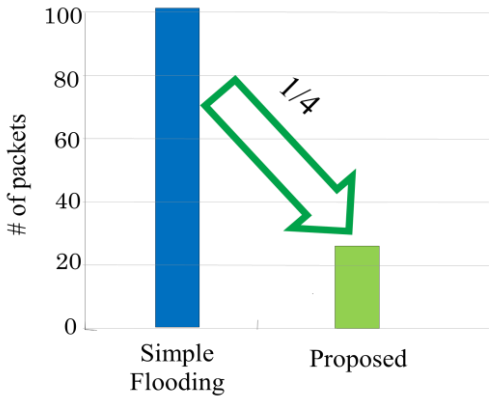


Fig.2 Effect of GPS based discovery

2.3 Load-aware broadcast-type contents delivery

In emergency situation (phase 1), a number of specific nodes may send urgent audio or video messages, such as evacuation instructions for disaster victims or instructions for rescue teams, to the entire network at a high bit rate without using any routing protocol. Flooding is the only delivery method to achieve this. However, as described in 3.2, conventional SF is not well suited for this purpose because it results in broadcasting too many unnecessary packets, which increases the chance of data collisions and buffer overflows, resulting in an increase in packet loss. Nodes in an ad hoc network are generally powered by batteries with a finite capacity and so reducing power consumption is an important issue. Since the transmission of redundant packets accelerates the consumption of battery power, an effective network protocol-based solution to reducing power consumption is to reduce the transmission of redundant packets.

Hence, we have proposed an effective data broadcasting, "Load-aware flooding" [6][7]. Our method uses the number of packets waiting for transmission in the MAC output queue of each node to obtain neighboring load information without periodical transmission of Hello messages. The reason why our method uses the number of packets in the queue is that at the time when the packets exist in a node's queue, the node is going to transmit frames exceeding its link capacity. If the node rebroadcasts a message in this situation, frame loss and collision due to buffer overflow are likely to happen. Therefore, the number of packets in the queue is useful information to recognize the load conditions without any status informing packet transmission.

By the computer simulation, it is clarified that this method reduces to 15% of simple flooding in best case for both the number of packets transmitted/received and the power consumed in the entire network (Fig.3). Our method also increases packets reachability from 71% to 99.9%, which means most of packets penetrate all over the network.

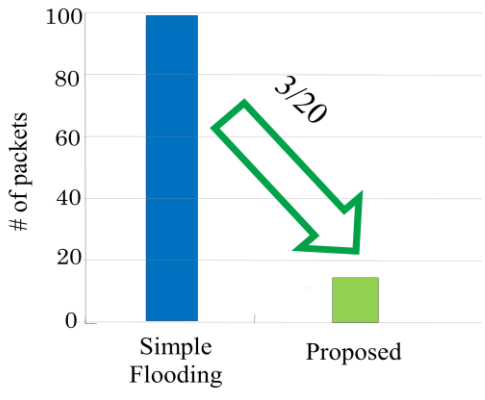


Fig.3 Effect of Load-aware Flooding

2.4 Effective secure communication

After contents discovery and routing table construction, the network enters stable phase 2. In phase 2, secure communication may be needed also in the MANET. The main problem of any public key based security system is to make each user's public key available to others in such a way that its authenticity is verifiable. In mobile ad hoc networks, this problem becomes even more difficult to solve because of the absence of centralized services, and possible network partitions. More precisely, two users willing to authenticate each other are likely to have access only to a subset of nodes of the network. The best known approach to the public key management problem is based on PKI (public Key Infrastructure) [8]. A public key certificate is a data structure in which a public key is bound to an identity by the digital signature of the issuer of the certificate. However, in a MANET without any infrastructure support, most traditional solutions are not directly applicable. The goal of our research is to understand the specific challenges for providing key management in MANETs and use this understanding to design an effective key management framework.

We, for the purpose, proposed a Trust Relationship based Self-Organize Key Management system [9][10] that allows users to create, store, distribute, and revoke their public keys without the help of any trusted authority or fixed server. Web-of-Trust fits naturally with MANETs, relying on each mobile node to issue certificates to other nodes at their own discretion [11]. This approach, however, suffers from frequent communication and much memory spaces because it must collect all the public key certificates beforehand. In order to resolve this defect, our approach, instead of collecting certificates themselves, generates and modifies "trust relationship lists" describing the trust relationship only without certificates among users.

By computer simulation, our proposal drastically reduces number of packets (1/100 to the existing method) (Fig.4).

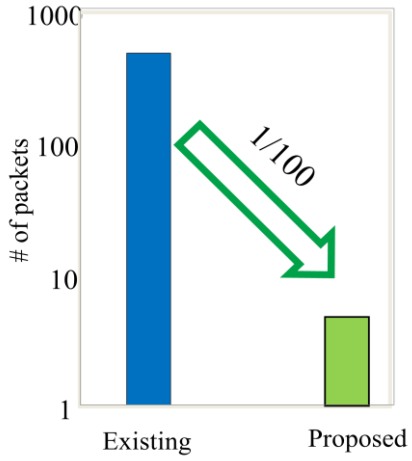


Fig.6 Effect of Trust Relationship based Self-Organize Key Management

3 Integrated evaluation

As shown in the previous chapter, our proposing methodologies are effective to reduce redundant packets that results in reduction of power consumption. In this chapter, we will evaluate what methodologies are used in a real situation and to what extent integrated three methodologies are effective.

3.1 What methodologies are used in real situation

In section 2.1, we have defined two different phases after the disaster happened, that is Phase 1: transient and Phase 2: stable. In those phases, what kind of communication is needed? And what kind of communication is executable? The purpose of network is of course communication. The MANET that might be established after the disaster is also aiming at communication.

3.1.1 Phase 1

Suppose emergency situation happens and neither communication nor power infrastructure becomes out-of-order. It is very difficult to realize one-to-one-communication. It is because as shown in 2.2, in Phase 1, IP addresses are not known to others. But in this phase you probably need communication to get information from and to tell your situation to other people. The former is "information discovery" related to 2.3 and the latter is information distribution (broadcasting) related to 2.3. Fortunately, these communications can be executed without any specific addresses but with "broadcast" address only.

3.1.2 Phase 2

After appropriate time, the MANET will be stable. What is stable means that each node has his own routing table and is able to make a point-to-point (p-p) communication. In this phase, normal p-p, ciphered p-p related to 2.4, and broadcasting communications will be needed. As for the discovery, p-p communication may cover the need.

3.2 Effect of methodologies when combined

Here we evaluate the integrated effect of our proposal in a real traffic condition according to the investigation in 3.1. Assumed parameters and values are as follows. Values are used and got in the simulation. Here, traffic ratio means ratio of how many users try the methodology to number of all the users during the observation time frame.

Table 1 Parameters

name of methodologies	items	parameters/values
common	number of network nodes	100
	average # of hops between nodes	3
discovery	traffic ratio	Rd
	#of packets sent (SF) for 1 trial	100
	# of packets sent (proposal) for 1 trial	25
	# of response packets	5
broadcast	traffic ratio	Rb
	# of packets (SF) for 1 trial	35,000
	# of packets (LDCF) for 1 trial	5,000
p-p	data traffic ratio	Rp
	traffic ratio of OLSR hello	0.1
	# of packets per communication	5
secure communication	data traffic ratio (ciphered)	Rc

3.2.1 Phase 1

As discussed in 3.1.1, in this phase, Discovery and Broadcasting are the main methodologies used.

(1) Discovery

100*Rd nodes initiate discovery. Simple Flooding discovery needs 100 packets for one discovery but our proposal needs 25. After discovery, discovered node responds to the initiator with 5 packets (same number as normal p-p data) and those make 3 hops. So, the average number of each methodology for discovery produces following numbers of packets.

SF: $100 \times R_d \times (100 + 5 \times 3)$ [packets] (1)

Our proposal: $100 \times R_d \times (25 + 5 \times 3)$ [packets] (2)

(2) Broadcasting

$100 \times R_b$ nodes broadcast information. Simple flooding case delivers 35,000 packet in average for one broadcast. Our proposal (LDCF) sends 5,000 packets.

SF: $100 \times R_b \times 35{,}000$ [packets] (3)

Our proposal: $100 \times R_b \times 5{,}000$ [packets] (4)

(3) Packet reduction effect in phase 1 by our proposal

By equations (1) through (4), the average number of transmitted packets for existing and our proposed methods will be;

Existing (SF): $11{,}500 \times R_d + 3{,}500{,}000 \times R_b$ [packets] (5)

Our proposal: $4{,}000 \times R_d + 500{,}000 \times R_b$ [packets] (6)

3.2.2 Phase2

As described in 3.1.2, in this phase 2, normal p-p (ciphered and unciphered), ciphered p-p, and broadcasting communications will be needed.

(1) normal p-p data (ciphered + unciphered)

$100 \times R_p$ nodes make p-p data communications each of which contains 5 packets in average and travels through 3 hops. Here we assume all the nodes use OLSR [12] as a routing protocol. Then average number of packets related to normal p-p data communication is:

$$100 \times R_p \times 5 \times 3 + 100 \times 0.1 \times 1 \text{ [packets]} \quad (7)$$

(2) Secure communication (ciphered)

$100 \times R_c$ nodes make secure communication. In the existing method, all the nodes must exchange their own certificates beforehand. Each exchange is made by broadcasting so that one certificate must be broadcasted 100 times. On the contrary, our method exchanges certificate only when normal p-p data communication happens. So, assuming number of packets to send a certificate is 2, average number of packets needed to ciphered communication becomes as follows.

Existing: $100 \times 100 \times 2 = 20000$ [packets] (8)

Our proposal: $100 \times R_c \times 3(\text{hop}) \times 2(\text{round trip}) \times 2$ [packets] (9)

(3) Broadcasting

This is same as equations (3) and (4).

(4) Packet reduction effect in phase 2 by our proposal

By equations (7), (8), (9), (3) and (4), the average number of transmitted packets for existing and our proposed methods will be;

Existing:

$$1{,}500 \times R_p + 20{,}000 + 3{,}500{,}000 \times R_b \text{ [packets]} \quad (10)$$

Proposal:

$$1{,}500 \times R_p + 1200 \times R_c + 500{,}000 \times R_b \text{ [packets]} \quad (11)$$

3.2.3 Evaluation

Here, we assume actual number for traffic ratio and evaluates the effect of our proposal specifically.

(1) Assumption (phase 1)

- Rd: 0.05 (about 5 users will make discovery)

- Rb: 0.05 (about 5 users will make broadcast)

Then, equation (5) becomes 175575 and (6) does 25200, so that reduction ratio is **14%**.

(2) Assumption (phase 2)

Rp: 0.5 (about 50 users will make p-p communication)

Rc: 0.01 (about 1 user will make discovery)

Rb: 0.01 (about 1 user will make discovery)

Then, (10) becomes 55750 and (11) becomes 5752, so that reduction ratio is **10**%.

These are just an example value but anyhow, in both phases 1 and 2, dominant traffic is broadcasting. In phase 2, however, the reduction effect of our proposal for secure communication is a kind of large, so that total reduction ratio is less than that in phase 1.

4 Conclusion

This paper summarizes achievement of ULPDDNS project for the network layer. It explained our proposed methodologies to reduce redundant packets for information discovery, broadcasting, and secure communication. It also shows effectiveness of each proposal. And it evaluates the integrated effects of those three proposal and shows our proposals drastically reduces redundant packets, that is , the reduction ratio becomes around 10% under some assumption.

From now, to finalize our project, we will make easy-to-understand explanation and deploy fruitful results to coming implementation oriented projects.

5 Acknowledgment

The ULP-DDNS project are partially supported by Core Research for Evolutional Science and Technology (CREST), Japan Science and Technology Agency.

6 References

[1] Subir Kumar Sarker, T.G. Basavaraju, and C. Puttamadappa, "Ad Hoc Mobile Wireless Network, Principles, Protocols, and Applications," Auerbach Publications, 2007.

[2] C.K. Prahalad, "The Fortune at the Bottom of the Pyramid: Eradicating Poverty Through Profits," Pearson Prentice Hall, 2009.

[3] Hiroaki Nishikawa, Kazuhiro Aoki, Hiroshi Ishii, and Makoto Iwata, "Intermediate Achievement of Ultra-Low-Power Data-Driven Networking System: ULP-DDNS", the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11) , PP.421-427, Jul. 2011.

[4] Naohide Fukushi, Keisuke Utsu and Hiroshi Ishii, "Information Discovery Mechanism using GPS over MANET", Proceedings of the 2009 International Conferences on Parallel and Distributed Processing Techniques and Applications (PDPTA'09), Jul. 2009.

[5] Keisuke Utsu, Naohide Fukushi and Hiroshi Ishii, "A Query-based Information Discovery method using Location Coordinates and its Contribution to Reducing Power Consumption in an Ad Hoc Network," the 2010 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10), Jul. 2010.

[6] Keisuke Utsu, Hiroaki Nishikawa and Hiroshi Ishii, "Load-aware Effective Flooding over Ad Hoc Networks", the 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09), Jul. 2009

[7] Keisuke Utsu and Hiroshi Ishii, "Load-aware Flooding over Ad Hoc Networks enabling High Message Reachability and Traffic," The 5th International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2010), pp.164-165, Apr. 2010

[8] R. Housley, W. Polk, W. Ford and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", IETF RFC3280, Apr. 2002.

[9] Hideaki Kawabata, Yoshiko Sueda, Osamu Mizuno, Hiroaki Nishikawa and Hiroshi Ishii, "Self-Organized Key Management Based on Trust Relationship List ",ICIN 2008 poster session, Oct. 2008.

[10] Hideaki Kawabata and Hiroshi Ishii, "Evaluation of Self-Organizing Key Management Framework Based on Trust Relationship Lists", the 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09), Las Vegas, USA, Jul. 2009.

[11] Srdjan Capkun, Levente Buttyan, and Jean-Pierre Hubaux: "Self-Organized Public-Key Management for Mobile Ad Hoc Networks", IEEE Transactions on Mobile Computing, vol.2, No.2, pp52-64, Jan-Mar 2003.

[12] T. Clausen, P. Jacquet, "Optimized Link State Routing Protocol (OLSR)", Internet Engineering Task Force(IETF), RFC3626.

Performance Evaluation of Load and Battery Charge Oriented Broadcast Streaming Method over Ad Hoc Networks

Keisuke UTSU[1], **Hiroaki NISHIKAWA**[2], **and Hiroshi ISHII**[1]
[1]Department of Communication and Network Engineering,
School of Information and Telecommunication Engineering, Tokai University, Minato, Tokyo, Japan
[2]Department of Computer Science, Graduate School of Systems and Information Engineering,
University of Tsukuba, Tsukuba, Ibaraki, Japan

Abstract – *Ad hoc networks are being studied as being resilient in a disaster situation. This paper considers cases where a specific number of nodes in an ad hoc network broadcast video and audio streams to the entire network. For this purpose, we have already proposed Load and Battery Charge Oriented Flooding (LBF) which reduces degradation in delivery quality, and energy consumption, and prevents nodes from being interrupted due to the complete discharge of their batteries. This paper shows performance evaluation of our method through the network simulation and confirms that LBF can prolong a network life time without degradation of delivery quality.*

Keywords: Ad Hoc Network, Broadcast, Flooding, Battery, Low Power Consumption

1 Introduction

Large-scale disasters often disable existing communications infrastructures or render stable power supply to communications terminals or base stations difficult. It is therefore important to study networks that can operate with low power consumption. Ad hoc networks are being studied as being resilient in a disaster situation [1].

This paper considers cases where a specific number of nodes in an ad hoc network broadcast video and audio streams over the entire network. In a disaster, several numbers of specific nodes will send urgent audio or video messages, such as evacuation instructions for disaster victims or instructions for rescue teams, to the entire network at a high bit rate without using any routing protocol. Flooding is the only delivery method to achieve this. However, conventional Simple Flooding (SF) [2] is not well suited for this purpose because it results in broadcasting too many redundant packets, which increases the chance of data collisions and buffer overflows, resulting in an increase in packet loss. Therefore, it is necessary to study how to maintain sufficient packet reachability. There have been some proposals for revising Simple Flooding to reduce redundant re-broadcasts [3-5].

Nodes in an ad hoc network are generally powered by batteries with a finite capacity and so reducing power consumption is an important issue. When an ad hoc network delivers video and audio streaming data, which generates packets at a high rate, the battery charge of nodes will dissipate rapidly. As a result, many nodes will stop functioning and the network's delivery capacity will fall. It is imperative to reduce redundant transmissions of packets in order to reduce the power consumption of nodes. In addition, it is necessary to reduce the chances of nodes becoming inoperative due to the complete discharge of their batteries.

To sum up, if we are to broadcast streaming video and audio data in a disaster situation, it is necessary (1) to achieve high packet reachability and reduce the degradation of delivery quality, (2) to reduce the energy consumption of nodes to make effective use of their battery charge, and (3) to reduce the chances of nodes becoming inoperative due to the complete discharge of their batteries. To achieve broadcast streaming method satisfying above three requirements, we have already proposed Load and Battery Charge oriented Flooding (LBF) [6]. This paper shows the evaluation of basic performance of LBF through the network simulation and confirms that our method can improve energy efficiency maintaining packet reachability. In addition this paper shows the evaluation of a network running time and confirms that LBF can prolong network running time without degradation of delivery quality.

Section 2 explains existing methods and their problems. Section 3 describes the proposed method. Section 4 shows the effectiveness of the proposed method using network simulation in terms of energy efficiency and packet reachability. Section 5 shows the evaluation of the network running time. Section 6 gives the conclusions.

2 Existing methods and their problems

As mentioned in Section 1, Simple Flooding (SF) [2] is the most frequently used broadcast-type delivery. The basic operation of SF is as follows: A node that originates packets (initiator node) broadcasts them. These packets reach all the nodes that exist within the area covered by the radio wave transmitted by the initiator node. A node that has received one of these packets re-broadcasts it. This process is repeated by subsequent nodes until the packets reach all the nodes in the network. When it is applied to the streaming delivery of video and audio data, which generates packets at a

high rate, redundant re-broadcasts occur, increasing the chances of collisions and buffer overflows, and resulting in a considerable degradation in delivery quality.

2.1 Counter-based schemes

There have been several proposals to revise Simple Flooding in order to reduce redundant broadcasts [3-5]. A well-known revision of Simple Flooding that does not depend on a special device, such as a Global Positioning System (GPS), is a Counter-based Scheme [3-4].

In this scheme, a node determines whether to re-broadcast a packet on the basis of the number of times the same packet has been received. A packet is identified by the combination of the ID of the node that generated it and the packet sequence. The basic operation of this scheme is as follows. When a node receives a packet, it sets the counter for that packet to "1". If it receives the same packet again during an arbitrary pre-defined time (*decision_time*), "1" is added to the counter. When the counter value reaches the counter threshold (*c_threshold*), re-broadcasting of that packet is suspended. If the counter value has not reached the counter threshold (*c_threshold*) after an elapse of *decision_time*, the packet is re-broadcast.

It is to be noted that the performance of this scheme greatly depends on *c_threshold* [4]. In a network in which nodes are scattered sparsely, re-broadcasts are limited when *c_threshold* is small (e.g., 2), but packet reachability is reduced. In a network in which nodes are distributed densely, *c_threshold* does not affect packet reachability too much. It is desirable to set *c_threshold* to 3 or 4. It has been reported that if *c_threshold* is as large as 6, this scheme behaves much like Simple Flooding, and thus cannot reduce redundant re-broadcasts [7].

To solve the above problem, Adaptive Counter-based Scheme, which sets *c_ threshold* dynamically, has been proposed [8]. In this scheme, each node sends a Hello packet periodically. These packets enable a node to determine the number of its surrounding nodes. The node determines whether to re-broadcast a packet on the basis of this information. A problem with this scheme is that the periodic transmission of Hello packets consumes the battery charge of each node and the wireless resources of the network.

While these two schemes reduce redundant re-broadcasts, they do not take the remaining charge of node batteries into consideration. They make no attempt to reduce re-broadcasts by nodes with low remaining battery charge, thereby increasing the chance that such nodes become inoperative due to the complete discharge of their batteries.

2.2 Existing battery-aware flooding methods

There have been a few proposals that take the remaining charge of node batteries into consideration [7-8]. Koide et al. [7] proposed a flooding scheme that uses a routing protocol. It sets a delay time that is dependent on the remaining battery charge. If a node receives the same packet again within its delay time, it discards the packet. Kasamatsu et al. [10] proposed a scheme in which the delay time set for each node is dependent on the distance from neighboring node, which is obtained using GPS, and the remaining battery charge. A node that has received the same packet again within its delay time, discards the packet. This scheme operates in such a way that nodes with a low battery charge are less likely to be selected for the re-broadcasting of packets. The scheme presented in [7] assumes the use of a routing protocol for the propagation of messages, and does not assume applications that generate packets at a high rate, such as a streaming delivery of video and audio data. Nor has it been evaluated for such applications. The scheme presented in [8] assumes that each node has a GPS and thus can obtain its location and distance information. However, in a disaster, it cannot be ensured that the correct location information can be obtained using a GPS. Therefore, it is unclear whether these schemes can be applied to the situation described in Section 1.

3 Proposed method

3.1 Assumed network environment and requirements

This paper focuses on unidirectional live streaming delivery. The network nodes are mobile communications terminals that are not equipped with any special device, such as a GPS, and can communicate over an IEEE802.11-series wireless LAN. No QoS (Quality of Service) control is considered. Packets are sent on a best-effort basis using UDP/IP.

We consider the application of our method to the situation described in Section 1, and study how to meet the following three requirements for the streaming delivery of video and audio data, which generates packets at a high rate. (i) The redundant transmissions of packets should be reduced in order to reduce degradation in delivery quality due to collisions and power consumption by terminals. (ii) Re-broadcasts by nodes with a low remaining battery level should be avoided in order to reduce the chance of these nodes becoming inoperative due to complete discharge of their batteries. (iii) Packets for checking the network state, such as Hello packets, should not be used, in order to reduce the network load and power consumption by nodes.

3.2 Load and Battery Charge Oriented Flooding (LBF)

To meet the requirements listed in Section 3.1, We have already proposed a flooding method that is sensitive to the

remaining battery levels of nodes, the existence of packets on the MAC transmission queue, and uses a counter in determining whether a packet should be re-broadcast or not [6]. We call this scheme Load and Battery Charge Oriented Flooding (LBF). Requirement (i) is met as follows. Each node monitors the number of packets existing in its MAC transmission queue at certain intervals. Then, if there is at least one packet in the MAC transmission queue, the node judges it is in a high load condition, and abandons re-broadcast. In addition, as is used in existing counter-based schemes, LBF determines whether to re-broadcast packets on the basis of the number of times that the same packet has been received, thereby reducing the transmission of redundant packets. Requirements (ii) and (iii) are satisfied as follows. Existing counter-based schemes use a fixed value for *c_threshold*, and thus cannot operate in a way that is sensitive to the remaining battery level of each node. LBF sets *c_threshold* dynamically in such a way as to reduce re-broadcasts by nodes with a low battery level. This makes it unnecessary for each node to send monitoring packets to its surrounding nodes in order to learn about their remaining battery levels. Nodes can operate autonomously.

The specific operation of the proposed method is as follows: For each node, the user sets, in advance, the maximum value of the counter threshold (*max_c_threshold*), and defines the range of a value generated by Random(), a function that generates a random value. Each node monitors its remaining battery level at certain intervals (*get_interval*), and reflects the value obtained in a variable, *remain_battery*. Then, *c_threshold* is calculated using the following equation:

$$c\_threshold = \mathbf{ceil(}\ max\_c\_threshold * (remain\_battery / max\_battery)\mathbf{)} \quad (1)$$

where **ceil(**Real *x***)** returns the value of a real variable, *x*, rounded up to the nearest integer. The value set in *max_c_threshold* and the value set in *c_threshold*, which is determined according to the remaining battery level.

A node that receives a packet from a node that initiated the packet (initiator node) operates as follows:

1. The node that has received the packet sets the packet's counter (*counter*) to "1" if it had not received the same packet earlier.

2. The time (*decision_time*) for which the node waits before it determines whether to re-broadcast the received packet is determined as follows:

$$decision\_time = \text{Random}() * 2(max\_c\_threshold - c\_threshold) \quad (2)$$

3. The node waits during *decision_time*. If during this time it receives the same packet again, it adds "1" to *counter*.

Variables and parameters

○Variable: Integer *queue*
// The num. of packets at the MAC transmission queue.
○Variable: Real *remain_battery*
// The remaining battery level at the node.
○Variable: Real *max_battery*
// The maximum battery level at the node.
○Variable: Integer *c_threshold*
// The threshold value of the counter.
○Variable: Integer *max_c_threshold*
// The maximum value of *c_threshold*
○Variable: Integer *counter*
// The number of times that the same packet has been received.
○Parameter: Integer *get_interval*
// The time interval for getting the num. of packets at the MAC transmission queue and the remaining battery level.
○Procedure: getQueue()
// The function to get the num. of packets at the MAC transmission queue.
○Procedure: getBattery()
// The function to get the remaining battery level.

(a) The method of getting data on the remaining battery level
Executed at certain intervals (*get_interval*)

```
queue ← getQueue()
if (queue > 0)
 then   (b) is not executed (all re-broadcast is canceled)
 else   remain_battery ← getBattery()
        c_threshold
          ← ceil(max_c_threshold*(remain_battery /max_battery)).
end if
End.
```

(b) The receiving and rebroadcast procedure

```
Receiving a packet.
if (The same packet as one that the node has already received)
 then  END.
 else  counter ← 1.
       decision_time ← Random() * 2^(max_c_threshold - c_threshold) .
       while (decision_time)
                if (Receiving the same packet again)
                  then  counter++.
                end if
       end while
       if (counter >= c_threshold)
         then   Rebroadcast is cancelled.
         else   Rebroadcast the packet.
       end if
end if
End.
```

Fig.1 The operation of a node for LBF

4. If, at the expiry of *decision_time*, counter does not exceed *c_threshold*, the received packet is sent to the lower layer and re-broadcast.

4 Evaluation of basic performance

This section compares the basic performance of the proposed LBF with those of existing schemes, in the video streaming, using a network simulator, OPNET [9].

4.1 Simulation configurations

The simulation configuration is as follows: Two geographical network areas are considered for simulation. In Space A, the area is 1000 m x 600 m, representing a network in which nodes are densely distributed. In Space B, the area is 2000 m x 1200 m, representing a network in which nodes are sparsely distributed. There are 100 nodes in the network, of which two nodes originates the video streams. The MAC layer is IEEE802.11b. The data rate is 2Mbps. The transmission power is 0.005W. The MAC layer of the nodes is IEEE802.11b. The data rate is 2 Mbps. The transmitted power is 0.005W. The received power threshold above which packets can be received successfully is -85 dBm. Nodes are initially located at random. All nodes move at a speed of [0.00, 4.00] m/s according to the random waypoint model. This is intended to simulate the walking of humans. A node that has received a packet from the initiator node re-transmits it using one of three delivery methods: the existing Simple Flooding (SF), the existing Counter-based scheme using a fixed *c_threshold*, and the proposed method. The parameters used in each method are as shown in Table 1. Using different values of the fixed *c_threshold*, three cases are considered for the Counter-based scheme: C4, C3 and C2. Similarly, using different values of *max_c_threshold*, three cases are considered for the proposed method: LB4, LB3 and LB2. In LBF, the interval at which the remaining battery level is monitored (*get_interval*) is 1 seconds.

Specific details of the video streams used are as follows: The use of H. 264 codecs [10] is assumed. The initiator node generates video streams as follows. On the "highway" [11] that uses the Quarter Common Intermediate Format (QCIF), 1,000 frames are encoded using jm14.2 [10]. The frame rate is 30 frames/s (i.e., a frame is generated every 33 ms). There are I frames and P frames and the group of pictures (GoP) is 10. Figure 2 shows the packet size distribution for 1,000 initiated packets. The initiator node repeats the transmission of these 1,000 frames 54 times, so generating 54,000 packets, which is equivalent to a 30-minute-long video stream. To take account of the rate at which packets are generated, the value generated by Random() at each node is set to [0, 33] ms.

The remaining battery level of a node is simulated as follows. The maximum battery level (*max_battery)* is 500 W*s. At the start of the simulation, the initial battery level of the initiator node is 500W*s (i.e., 100% full), and those of other nodes are random in the range [100, 400] W*s (i.e., 20% to 80% full).

Table 1 Parameters for each delivery method

	c_threshold	*max_c_threshold*
SF	-	-
C4	4 (fixed)	-
C3	3 (fixed)	-
C2	2 (fixed)	-
LB4	Determined by Eq.(1)	4
LB3	Determined by Eq.(1)	3
LB2	Determined by Eq.(1)	2

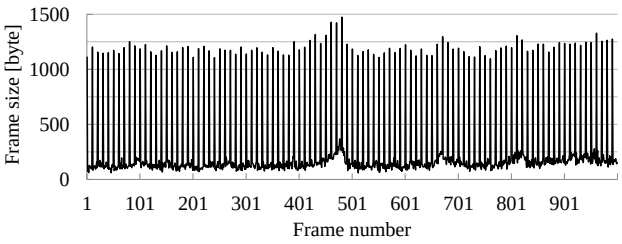


Fig.2 Frame size distribution of generated video

Feeney et al. [12] have indicated that the transmission power, *tp* [μW*sec], and the reception power, *rp* [μW*sec], for a single packet are as follows:

$$tp = 2.000 * \textit{frame length}\ [\text{byte}] + 270 \quad (3)$$

$$rp = 0.500 * \textit{frame length}\ [\text{byte}] + 60 \quad (4)$$

We assume that the above power is consumed by a node each time it sends or receives a packet. Any power that may be consumed while no packet is being sent or received is disregarded. When the remaining battery level of a node is zero, the node ceases to operate.

4.2 Evaluation items

The simulator executes 10 trials for each random seed. The average for all the trials is calculated for each of the following evaluated items.

(i) Average remaining battery level at the end of the streaming [W*s]

This is the average remaining battery level at the end of the streaming for all the nodes in the network. The larger this value, the better.

(ii) Average energy consumed by a node till the end of the streaming [W*s]

This is the average energy consumption per node from the start until the end of the streaming for all the nodes in the network. The smaller this value, the better.

(iii) Percentage of nodes whose operation was stopped due to complete battery discharge [%]

This is the percentage of the nodes that had ceased to operate by the end of the streaming among all the nodes in the network. The smaller this value, the better.

(iv) Average packet reachability [%]

Packet reachability is the percentage of the nodes that have received successfully a packet generated by the initiator node among all the nodes in the network. The average packet reachability is the average for all the packets generated by the initiator node. The larger this value, the better.

4.3 Evaluation results

The evaluation result is shown in Fig. 3.

(i) Average remaining battery level at the end of the streaming

LB2 to LB4 registered higher values than any existing scheme, in both Spaces A and B. In particular, LB2 produced the highest value in both Spaces A and B. These results indicate that the proposed scheme can preserve a higher battery level than the existing schemes.

(ii) Average energy consumed by a node till the end of the streaming

In both Spaces A and B, LB2 to LB4 consumed less energy than the existing schemes. In particular, LB2 consumed the least power, 29.6% down from SF, and 33.4% down from C2 in Space A. 55.4% down from SF and 42.9% down from C2 in Space B.

(iii) Percentage of nodes whose operation was stopped due to complete battery discharge

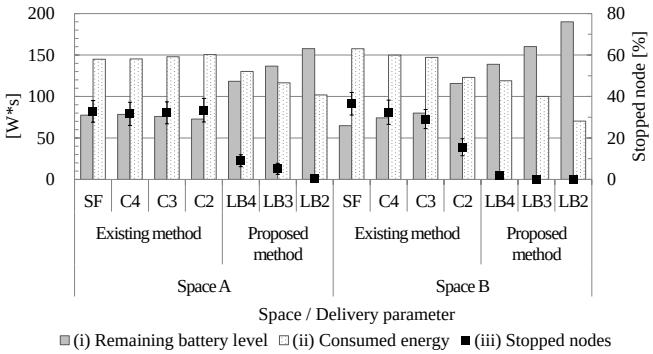
The deviation bars show the (average +/- standard deviation). LB4 to LB2 produced a lower percentage than any existing scheme in both Space A and B. These results indicate that the proposed method results in fewer cases in which nodes become inoperative due to complete discharge of their batteries than the existing schemes.

(iv) Average packet reachability [%]

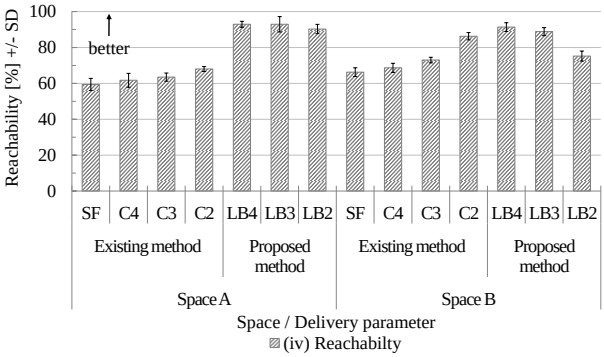
The deviation bars show the (average +/- standard deviation). LB4 and LB3 produced higher average packet reachability than any existing scheme in both Spaces A and B. In Space B, the average packet reachability value of LB2 is lower than that of C2, which registered the highest value among the existing schemes.

4.4 Discussion

The simulation results described in the above section allow us to conclude the following. In comparison to existing methods, the proposed method can (i) increase the average remaining battery level at the end of streaming, (ii) reduce the average energy consumed by nodes from the start to the end of the streaming, and (iii) reduce the percentage of nodes which became inoperative due to complete battery discharge. These affects are due to LBF's mechanism of dynamically setting *c_threshold* according to the remaining battery level of each node -- in particular of avoiding cases when nodes with a low battery level re-broadcast packets. LB2 produced the best performance in these three respects, (i), (ii), and (iii). The delivery quality was evaluated in terms of (iv) the average packet reachability. In networks where nodes are distributed densely (Space A), the proposed method produced higher average packet reachability than Simple Flooding or Counter-base schemes with fixed *c_threshold* (C4-C2). In networks where nodes are distributed sparsely (Space B), LB2 performed the lower packet reachability than C3, LB4 and LB3. To maintain packet reachability at that time, *max_c_threshold* should not be too small value such as 2. On the other hand, packet reachability for LB4 and LB3 are greater than SF and C4-C2. Therefore, LB4 and LB3 should be applied to the delivery parameter in the proposed method.



a. (i) Remaining battery level, (ii) Consumed energy, and (iii) % of Stopped nodes

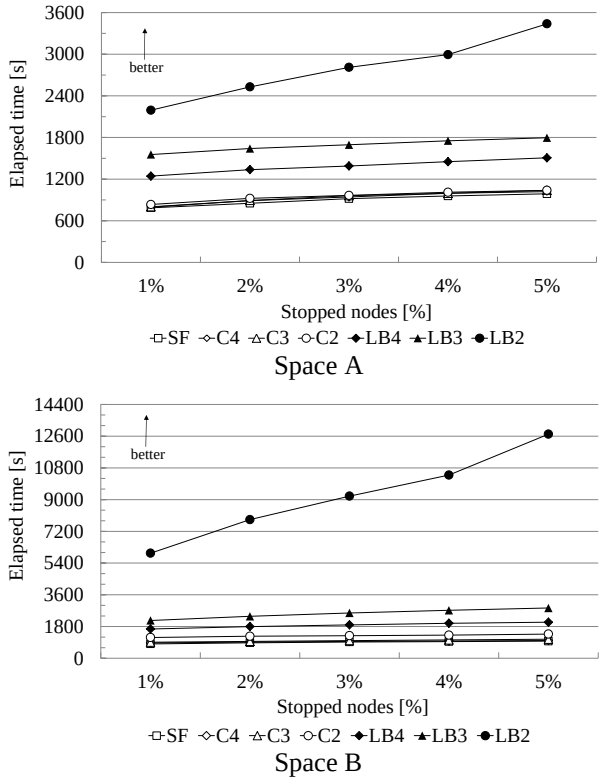


b. (iv) Average Packet Reachability

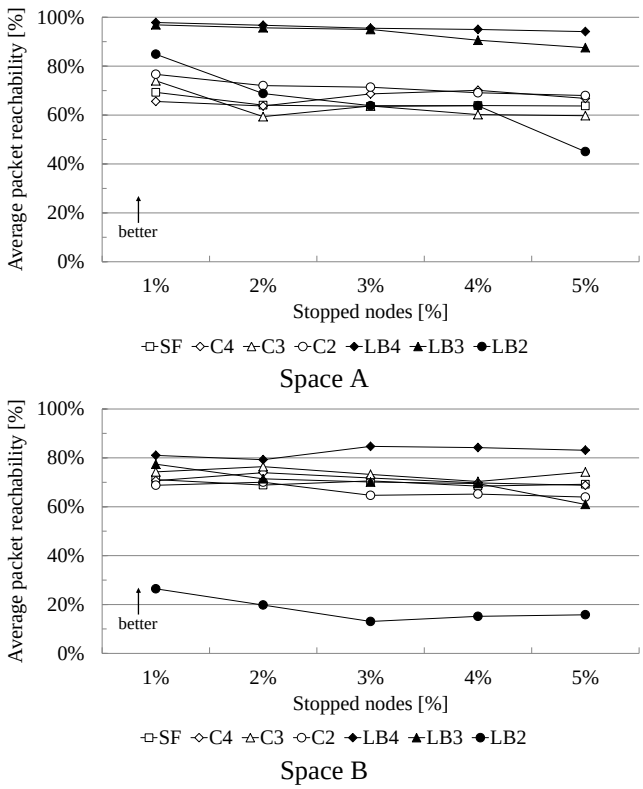
Fig. 3 Simulation result of basic performance for each set of delivery parameters

5 Evaluation of network runninng time and packet reachability

To evaluate the proposed method can prolong network lifetime without degradation of delivery quality, in this section,



a. (v) % of stopped nodes against elapsed time



b. (vi)Average packet reachability against % of stopped nodes

Fig. 4 Simulation result of network life time and delivery quality for each set of delivery parameters

we evaluate the network running time, which is the elapsed time when specific percentage of nodes stopped due to complete battery discharge, and packet reachability.

5.1 Simulation configurations

The simulation configuration is the same in Section 4.1. Each initiator node generates 1,000 frames repeatedly based on the frame size distribution which is shown in Fig.2.

5.2 Evaluation items

The simulator executes 10 trials for each random seed. The average for all the trials is calculated for each of the following evaluated items.

(v) % of stopped nodes against elapsed time

The elapsed times when specific percentages (1-5%) of nodes stopped ware evaluated. The longer time, the better.

(vi) Average packet reachability against % of stopped nodes

The percentage of the nodes that have received successfully a packet generated by the initiator node among all the nodes in the network was evaluated at the time when 1-5 percentages of nodes stopped. The larger this value, the better.

5.3 Evaluation results

The evaluation result is shown in Fig. 4.

(v) % of stopped nodes against elapsed time

LB2 to LB4 performed the longer running time than the existing methods, in both Space A and B. In particular, LB2 performed the longest running time. These results indicate that our methods can prolong the network lifetime.

(vi) Average packet reachability against % of stopped nodes

The result in Fig.4b shows that LB4 performed the highest value regardless of the percentage of stopped nodes. In other words, LB4 is the best in terms of packet reachability. Considering the both results Fig.3b and Fig.4b, the packet reachability for LB4 and LB3 are greater than or equals to that for the existing methods at the time when several nodes stop. This means that LB4 and LB3 can maintain the packet reachability whether several nodes stop or not. On the other hand, as shown in Fig.3b, the packet reachability for LB2 was not as high as that for LB4 and LB3. On the contrary, Fig.4b

indicats that LB2 performs extremely low packet reachability at the time when several nodes stops.

5.4 Discussion

The simulation results described in the above section allow us to conclude the following. In comparison to existing methods, the proposed method (LB2-LB4) can prolong the network running time. LB2 which sets the lowest *max_c_threshold* value performed the longest running time. However, considering the case where several nodes stop due to the battery discharging, LB2 performed the lower packet reachability than any other delivery parameters. To maintain packet reachability at that time, *max_c_threshold* should not be too small value such as 2. On the other hand, packet reachability for LB4 and LB3 are greater than or equal to SF and C4-C2. Therefore LB4 and LB3 should be applied to the delivery parameter in the proposed method.

6 Conclusions

This paper has considered broadcast streaming in ad hoc networks which can achieve to maintain delivery quality and long network lifetime. For this purpose, we have already proposed Load and Battery Charge Oriented Flooding (LBF), in which re-broadcast at the loaded nodes are canceled and the counter threshold used to determine whether a packet should be re-broadcast or not, *c_threshold*, is dynamically set according to the remaining battery level of each node. In this paper, the basic performance of the proposed method has been evaluated using the network simulation. The evaluation results have shown that, in comparison to existing flooding schemes, the proposed method can reduce energy consumption by nodes, reduce the chances of nodes becoming inoperative due to battery complete discharge of node batteries, and avoid degradation of delivery performance. This paper also evaluated the network running time and the packet reachability. The evaluation results have shown that, in comparison to existing flooding schemes, LB4 and LB3 in the proposed method can prolong the network running time without degradation of delivery performance. In future, we will evaluate the QoS of played-back video and audio streams and implement on communication devices.

7 Acknowledgments

This research has been supported by Core Research for Evolutional Science and Technology (CREST), Japan Science and Technology Agency (JST), and KAKENHI (Grant-in-Aid for JSPS Research Fellows), Japan Society Promotion of Science (JSPS).

8 References

[1] C. Siva Ram Murthy and B. S. Manoj, "Ad Hoc Wireless Networks - Architectures and Protocols," Prentice Hall, Professional Technical Reference

[2] Jorjeta G. Jetcheva, David A. Malts, "A Simple protocol for Multicast and Broadcast in Mobile Ad Hoc Networks", IETF MANET Working Group Internet-Draft, <draft-ietf-manet-simple-mbcast. txt>, 2001

[3] Brad Williams, Tracy Camp, "Comparison of Broadcasting Techniques for Mobile Ad Hoc Networks", Proceedings of the 3rd ACM International Symposium on Mobile Ad Hoc Networking and Computing, pp. 194-205, 2002

[4] Yu-Chee Tseng, Sze-Yao Ni, Yuh-Shyan Chen, Jang-Pinig-Sheu, "The Broadcast Problem in a Mobile Ad Hoc Network", Wireless Networks Volume 8, Springer, pp. 153-167, Kluwer Academic Publishers, 2002

[5] Yu-Chee Tseng, Sze-Yao Ni, En-Yu Shih, "Adaptive Approaches to Relieving Broadcast Storm in a Wireless Multihop Mobile Ad Hoc Network", IEEE Transactions on Computers, Vol. 52, No. 5, pp. 545-556, 2003.

[6] Keisuke Utsu, Hiroshi Ishii, "Load and Batttery Charge oriented Flooding for Broadcast Streaming over Ad Hoc Networks", IEEJ Transactions on Electronics, Informations and Systems, Vol.132, No.5, pp.640-648, Mar 2012

[7] Toshio Koide, Hitoshi Watanabe, "A Versatile Broadcasting Algorithm on Multi-Hop Wireless Networks: WDD Algorithm", IEICE Trans. Fundamentals, Vol. E87-A, No. 6, pp. 1599-1611, Jun. 2004

[8] Daisuke Kasamatsu, Norihiko Shinomiya, and Tadashi Ohta, "A Broadcasting Method Considering Battery Lifetime and Distance between Nodes in MANET", IEICE Trans. Commun. , Vol. J91-B, No. 4, pp. 364-372, 2008

[9] The network simulator "OPNET", http://www.opnet.com

[10] H. 264/AVC Reference Software (jm14. 2), http://iphome. hhi. de/suehring/tml/download/old_jm/

[11] Patrick Seeling, Frank H. P. Firzek and Martin Reosslein, "Video Traces for Network Performance Evaluation," Springer, 2007

[12] Feeney L. M., Nilsson M., "Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment", INFOCOM 2001, Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies, Vol. 3, pp. 1548-1557, IEEE, 2001

Power-Performance Verification of Ultra-Low-Power Data-Driven Networking Processor: ULP-CUE

Shuji SANNOMIYA[1], **Kazuhiro AOKI**[2], **Makoto IWATA**[3], **and Hiroaki NISHIKAWA**[1]
[1]Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba Science City, Ibaraki, 305-8573 Japan
[2]Information Infrastructure Laboratory, Inc., Tsukuba Science City, Ibaraki, 305-0003 Japan
[3]School of Information, Kochi University of Technology, Kami, Kochi, 782-8502 Japan

Abstract— *Ultra-low-power protocol handling is indispensable to realize and sustain the communication in ad hoc network established by battery-operated devices, especially in emergent situations. To realize the ultra-low-power, real-time multiprocessing essential for the protocol handling should be realized with essential power consumption in which power consumption is confined into only target program execution. In this paper, a data-driven processor, named ULP-CUE, is proposed to realize the real-time multiprocessing with the essential power consumption. The ULP-CUE is fully realized by self-timed elastic pipeline which consumes power only for circuits contributing processing as a result of localized and exclusive data transfer, and its instruction execution pipeline is designed to transfer data to only stages essential for executing instructions. The power-performance evaluation with a prototype LSI shows that the proposed ULP-CUE can work with approximately 40% of energy consumption compared to the conventional implementations of the real-time multiprocessing.*

Keywords: data-driven processor, protocol handling, real-time multiprocessing, self-timed pipeline

1. Introduction

Low-power communication protocol handling is indispensable to enhance the sustainability of communication, especially in ad-hoc networks of mobile devices linked by wireless communication schemes such as MANET [1]. To ensure the communication, processors should execute required communication protocol handling in real-time for satisfying both turn-around and response times required, and also realize multi-processing dealing with multiple streams input into the communication protocol handling as long as their processing capability is available in order to satisfy a given throughput.

To realize such real-time multiprocessing without any extrinsic overheads on program execution, a data-driven processor named CUE and its successors are already studied [2], [3], [4]. By fully utilizing the no-extrinsic-overhead real-time multiprocessing of the CUE, the authors conduct a collaborative research project, named ULP-DDNS, to realize an ultra-low-power networking system [5].

The CUE is a realization of data-driven processing scheme in which instruction execution is initiated on the arrival of input data. This passive instruction execution realizes the real-time multiprocessing without extrinsic controls such as context switching violating both the real-time constraint and power saving. The circuit of the CUE is realized only by self-timed elastic pipeline (STP) in which only pipeline stages with valid data drive their pipelined logics autonomously, i.e., the signal gating is provided naturally at pipeline stage level. Moreover, the empty stages without valid data can be powered-off by using a stage-by-stage power gating scheme to reduce leakage current through the empty stages [6].

However, the instruction execution pipeline in the originally-proposed CUE harms power-performance efficiency of the protocol handling. This is because the conventional pipeline executing instructions enforces every data to pass through the matching stage which detects the arrival of two operands for an instruction even though most of instructions in the protocol handling can be executed with single operand. To eliminate the redundant processing time and power dissipation due to the matching-centric instruction execution pipeline, ultra-low-power CUE (ULP-CUE) realized by a circular pipeline with a shortcut to bypass the matching stage is already being studied [7].

In this paper, the power-performance of the ULP-CUE is evaluated by using a prototype LSI chip of the ULP-CUE. To verify the essential power consumption of the ULP-CUE, the power consumption of a protocol handling is measured by using its evaluation board and it is compared with that of the conventional CUE. The rest of paper is organized as follows. Section 2 describes the requirement for a networking platform to realize the ultra-low-power protocol handling, and the architecture of the ULP-CUE is explained in Section 3. The power-performance verification is shown in Section 4, and then our conclusion is discussed in Section 5.

2. Requirement to realize ultra-low-power networking platform

To realize the real-time multiprocessing with ultra-low-power, the power should be consumed only for executing

instructions of target programs. This unique but absolutely necessary feature is called essential power consumption in this paper. The passive behavior realized by data-driven principle makes it possible to realize the essential power consumption.

This section describes how the real-time multiprocessing with essential power consumption is realized based on the data-driven principle.

2.1 Real-time multiprocessing with essential power consumption

The real-time multiprocessing can be achieved if the execution of each target process is initiated as soon as its input data arrive while the target processes are executed simultaneously and independently from each other in a processor. To realize the real-time multiprocessing with ultra-low-power, any overheads on process execution should be eliminated.

The simultaneous and independent process execution can be realized by either concurrent processing scheme or pipeline processing scheme. The former divides spatially a computational resource into multiple computational resources to execute one process on each computational resource, and the latter divides temporally a computation resource into multiple computational resources which are called stages to execute multiple processes at the different stages. In the networking platform to execute protocol handling, the execution order of the processes is undetermined before the execution because it depends on the arrival timing of communication packets, i.e., it depends on user's demands. This fact imposes the dynamic allocation of the computational resources on the processing schemes. To realize the dynamic allocation, all computational resources should be monitored to detect whether they are idle or not, and target processes are assigned to the idle resources in the concurrent processing scheme. In contrast, the dynamic allocation can be realized only by assigning the target processes to the first stage when the first stage is idle in the pipeline processing scheme, and this simplicity results in less power consumption; therefore the pipeline processing scheme is indispensable to realize the real-time multiprocessing with essential power consumption.

To realize the pipeline, the passive behavior of the data-driven principle is exploited for providing the essential power consumption. Self-timed elastic pipeline, which is abbreviated as STP in this paper, is a realization of the passive behavior of the data-driven principle in circuit implementation.

In the STP, only pipeline stages with valid data are driven exclusively, and thus power consumption is confined into the circuit essential to execute the target processes. This exclusive driving is realized as a result of the local negotiation between adjacent stages. Figure 1 shows the basic structure of the STP whose each stage consists of a

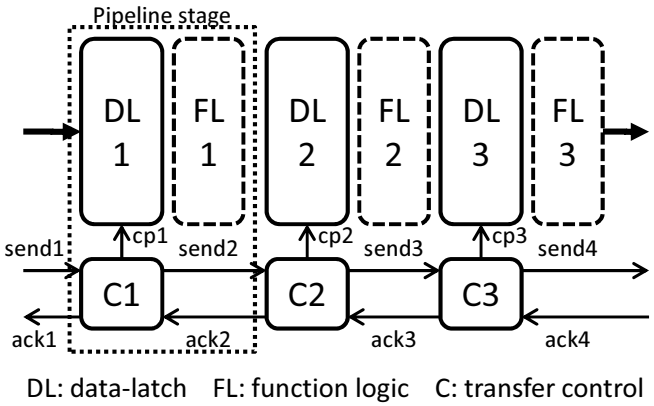


Fig. 1: Self-timed elastic pipeline.

data-latch (DL), functional logic (FL) and transfer control unit (C). The STP is a kind of asynchronous bundled-data pipelines, and the local negotiation is realized by a four-phased handshake [8]. The valid data in the STP are transferred between adjacent stages as follows.

- Reset: After the assertion of the reset signal, the C negates both its send signal representing transfer request and ack signal representing acknowledge.
- The C asserts its ack signal after its send signal is asserted.
- After the assertion of the ack signal, the preceding C negates its send signal.
- After the negation of the send signal, the C asserts both its gate open signal and its send signal and negates concurrently its ack signal, only if the ack signal is negated. As a result, the token is latched in the stage to which the C belongs.
- The succeeding C repeats the above steps similarly to the C.

This localized data transfer naturally results in the signal gating at the stage level, and thus the STP confines the power consumption by transistor switching into only the stages executing the target processes. Moreover, the stages in the STP can be powered off only when they are empty, i.e., they latch no valid data [6]. Therefore the essential power consumption can be realized by the STP.

2.2 Data-driven processing scheme for essential power consumption

In the real-time multiprocessing with the pipeline processing scheme, target processes should be divided into finer processes which can be executed concurrently, and the pipeline execution of each finer process should be initiated immediately after the process become executable, i.e., the input data for the process arrives. Figure 2 illustrates the real-time multiprocessing with pipeline processing scheme.

To keep the real-time processing of each target process, the target processes should be finely divided for sharing a

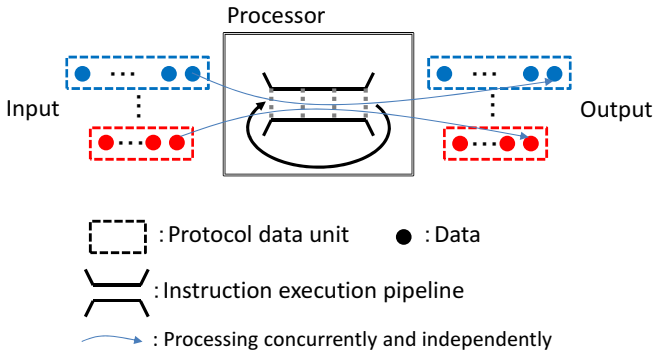


Fig. 2: Real-time multiprocessing.

pipeline among the target processes, and thus fine-grain parallelism inherent in the target processes should be exploited. That is, instruction level parallelism (ILP) which is the parallelism of instruction should be exploited exhaustively because instruction is the finest executable unit of processes.

The arrival timing of the input data depends essentially on the user's communication demands and thus it is unforeseen and determined at runtime. Therefore, the data-driven processing scheme in which instructions become executable only after their input data arrive is a natural processing scheme and thus should be introduced to realize the real-time processing with pipeline processing scheme.

Figure 3 illustrates how the programs are executed by the data-driven processing scheme. As illustrated in the figure 3, the programs are defined by data-flow graph (DFG). The DFG consists of nodes and arcs, and each node describes an instruction while each arc represents the data-dependency between two nodes. The data-dependencies between instructions represent naturally the ILP inherent in the programs, and thus describing target program by using DFG results in extracting the ILP in the target programs. To execute instructions concurrently at the different pipeline stages, the instruction execution logic is pipelined. In the figure 3, the first stage of the pipelined instruction execution logic is illustrated with the passage of time, by supposing the number of pipeline stages is 3 for ease of explanation, i.e., the execution of each instruction is completed after 3 ticks.

The figure 3 shows that the instructions are executed only after their input data arrive. Based on the data-driven processing scheme, the instructions with input data can be executed independently from each other, and thus the execution of each instruction is independent of that of the others in the same pipeline. Therefore, the instruction execution can be realized without any extrinsic control which stops and resumes the execution of instructions, and the power is consumed only for the instruction execution in principle.

In contrast, conventional non-data-driven processing schemes are essentially the extensions of von-Neumann type sequential processing scheme in which single program is executed in a pipeline, and they achieve real-time processing

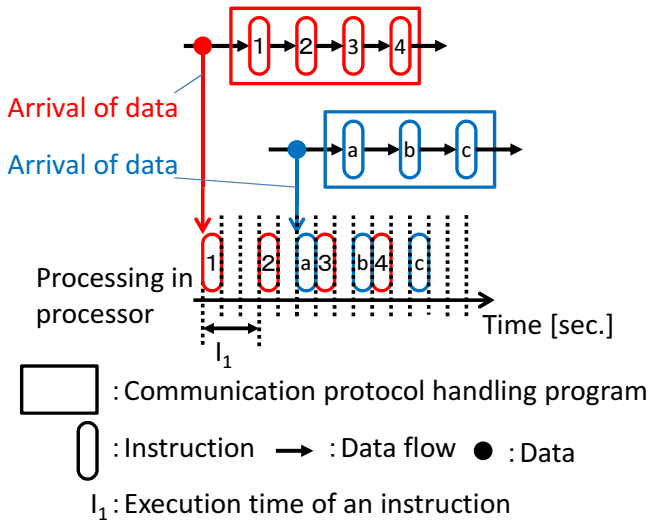


Fig. 3: Data-driven processing scheme for real-time multiprocessing.

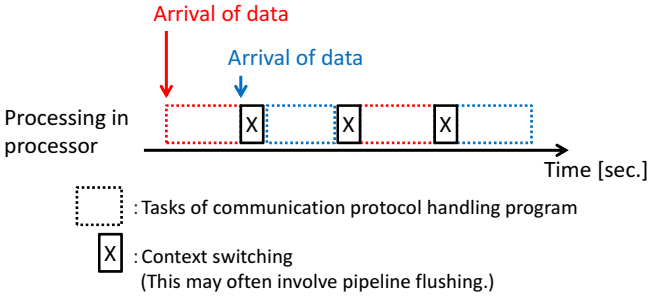


Fig. 4: Conventional quasi-multiprocessing scheme.

by switching divided executable process units called tasks in a pipeline. In such conventional quasi-multi-processing schemes, data transfer among instructions is realized by using a temporal memory mechanism called registers, and the content of the registers, called context, belongs to one program basically; therefore, the context of the currently executed program should be saved and then the context of another program to execute should be recovered into the registers. This saving and recovering of the context is called context-switching. Moreover, the pipeline should be flushed before the context-switching if instructions are being executed in the pipeline, and this pipeline flushing depends on the arrival timing of input data. That is, to switch the tasks, the context-switching is executed as illustrated in figure 4, and the power is consumed not only the instruction execution but also the context-switching and pipeline flushing. This fact violates obviously the essential power consumption.

Consequently, the data-driven processing scheme with the self-timed pipeline is necessary to realize the real-time processing with essential power consumption.

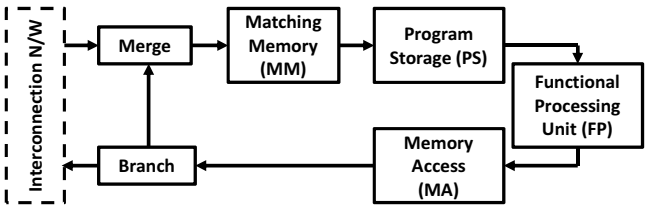


Fig. 5: Block diagram of CUE processors.

3. Ultra-low-power CUE

As described in the previous sections, the CUE and its successors, collectively called CUE processors in this paper, are already proposed as the implementations of the data-driven processing scheme with the STP. The CUE processors are designed by focusing on the performance efficiency within a given circuit area, and thus the essential power consumption is unrealizable by using them.

This section reveals how the CUE processors can be optimized for the essential power consumption, and presents an optimized CUE processor, named ULP-CUE.

3.1 Optimization of CUE processors

Figure 5 shows the functional block diagram of the CUE. The CUE processors represents directly the data flow of tracing the data-dependencies between instructions, and it consists of matching memory (MM), program storage (PS), functional processing unit (FP) and memory access (MA). As shown in the figure 5, the CUE processors are realized by a circular pipeline connecting the MM, PS, FP and MA by using merge and branch stages. The merge stage accepts tokens from two preceding stages in order of arrival and transfers the tokens to a succeeding stage while the branch stage transfers each token to one of two succeeding stages selectively.

In the CUE processors, each data is packetized with information required to execute instruction in order to process every data independently, i.e., instructions are executed independently from each other. This self-contained data is called token, and the contained information is called tag. The tag consists of operation code, destination node number and generation. The generation is the number used to identify the stream to which the data belongs and specify the order of the data in a stream. On the other hand, every instruction is assigned unique number which is called destination node number, and the destination node number is used to identify an instruction to which the data is input.

The MM, PS, FP and MA are used to execute an instruction according to the tag. The MM provides temporal storage to keep tokens whose operation code represents binary operation, until the arrival of the paired tokens, and it outputs either a token containing two operands for binary operation or a token containing single operand for unary operation. To realize the pairing of tokens, the MM is realized by a content-addressable memory (CAM) whose key consists of the generation and destination node number. The operand or a pair of operands in the token output from the MM is processed according to the operation code in the FP, and the FP outputs a token whose data is the result of the operation. After that, the operation code of the token is replaced with that of an instruction specified by the destination node number of the token in the PS storing instructions of the target programs.

In contrast to the original circular pipeline proposed in [2], the PS is placed between the MM and FP to reduce power dissipation. The MM outputs a token for binary operation after two tokens input to the binary operation arrive, i.e., the operation code of one of the tokens is discarded; therefore, the power dissipation to read the operation code from the PS can be eliminated if the PS is placed after the MM. This change has no effects on the function of the circular pipeline because the function of the PS is independent from that of the FP and MA.

The circular pipeline is commonly used to execute every instruction in order to reduce the circuit area which is rather limited in early sub-micron process technology era and is becoming inconsequential compared to power consumption in deep sub-micron and beyond era. As for the power consumption, the MM may occupy more than a half of the power consumption required to execute an instruction because the detection of the arrival of the tokens in the MM is typically realized by using the CAM in which the keys stored are thoroughly compared to the input key for every incoming token.

Fortunately, more than a half of the instructions of the protocol handling can be executed without driving the MM. Such instructions are classified into two types: single-operand instructions and single-operand with constant instructions. An example of the former is an instruction to realize increment/decrement operation and that of the latter is an instruction to read/write memory with absolute address. In this paper, these two types of instructions are collectively called single token instructions, because they can be executed after an input token arrives. Our previous implementation of the protocol handling reveals that the single token instructions occupy the most part of the DFG. For example, a DFG of UDP/IP handling shows that approximately 80% of the executed instructions is the single token instruction. Consequently, the circular pipeline of the CUE processors should be revised to realize the essential power consumption.

3.2 Optimized circular pipeline

To realize the essential power consumption, the circular pipeline to connect the MM, PS, FP and MA should be optimized. Obviously, the PS to fetch instructions and the FP to operate data are indispensable to execute every instruction. The PS and FP are used repeatedly to execute instructions

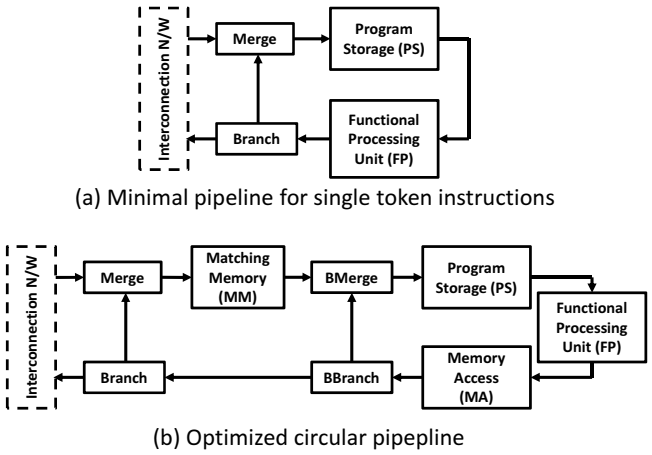


Fig. 6: Block diagram of ULP-CUE.

in programs by tracing the data-dependencies between the instructions. To realize this repetition directly and naturally, the PS and FP should be connected by using a circular pipeline as shown in figure 6(a).

To actualize the operation between two operands, the operand arrived first should be temporally stored until the other operand arrives. As described in the previous section, the real-time multiprocessing is realized by concurrent execution of multiple instructions. To make it possible to execute instructions concurrently, the operands arriving first should be stored separately for each instruction, and they should be drawn after companions arrive. The storing and drawing operands should be realized immediately after the operands arrive in order to avoid any delay on the instruction execution. This requirement leads to the introduction of the MM because the CAM of the MM can realize the dynamic allocation of memory space for newly arrived operands and all contents of the CAM can be searched concurrently and thus the storing and drawing can be realized without any delays.

Unfortunately, the MM realizes such fast storing and drawing at the expense of a large amount of the power consumption as described above. In contrast, memories realized by SRAM can store relatively large amount of data with less power consumption, and thus they are indispensable to store large size data. For example, the payload data of a packet should be buffered during header manipulation of the packet. In addition, they should store data commonly used among the target programs. The table of IP addresses is an example of such data. Moreover, the data determined before the program execution, such as constants or default values should be also stored into the SRAM-based memories. These facts lead to the instruction of the MA stage to realize the read/write of the SRAM-based memories.

In principle, the single token instructions can be executed without the MM, and thus the MM should be bypassed during the execution of them. On the other hand, the MA is required by not only the single token instructions because the result of the operation may be stored into the SRAM-based memory at least. To realize the bypass for the single token instructions, the circular pipeline shown in the figure 6(a) should be extended as illustrated in figure 6(b). Consequently, a data-driven processor realized by the optimized circular pipeline can execute the protocol handling with the essential power consumption. This data-driven processor is named ULP-CUE, which stands for ultra-low-power CUE, in this paper.

4. Evaluation on energy consumption

The ULP-CUE proposed in the previous section realizes the protocol handling with the essential power consumption. As a result, the ULP-CUE can reduce the energy and execution time of the protocol handling in comparison with the CUE processors because the single token instructions are executed without the MM in the ULP-CUE. The reduced execution time makes it possible to supply low-voltage resulting in less energy. To verify the power-performance efficiency of the ULP-CUE, the energy and execution time of the ULP-CUE is compared to that of the CUE processors, and the energy reduction is estimated based on a prototype LSI of the ULP-CUE.

As a target protocol handling, UDP/IP handling is focused on because its connection-less communication realizes streaming with less energy in comparison to TCP/IP handling and thus it is indispensable for the broadcast-type information transfer in the ad hoc networks especially in the emergent situations.

4.1 Prototype LSI

To verify the ULP-CUE experimentally, a prototype LSI housing four ULP-CUE's in a $4.2mm^2$ die is designed and fabricated by using a 65nm CMOS 7-metal-layer process. Figure 7 shows the layout of the ULP-CUE which is consisted of a 13-stage circular pipeline realizing the PS, FP, MA, MM, Merge and Branch. The ULP-CUE has an instruction set sufficient to execute the UDP/IP handling, and the function of the ULP-CUE is verified by checking that the outputs of the UDP/IP handling on the ULP-CUE are identical to the expectation values of the UDP/IP handling.

Although the prototype LSI's of the CUE processors are already fabricated to show the feasibility and effectiveness, they have different instruction sets and are fabricated by using different process and CAD tools. To eliminate these differences and clarify the difference derived from the circular pipeline, the energy and execution time of the CUE processors are estimated based on the measured results of the prototype LSI of the ULP-CUE.

The difference between the circular pipeline of the ULP-CUE and the CUE processors is a bypass realized by the BMerge and BBranch stages. To estimate the energy and execution time for the CUE processors, the energy

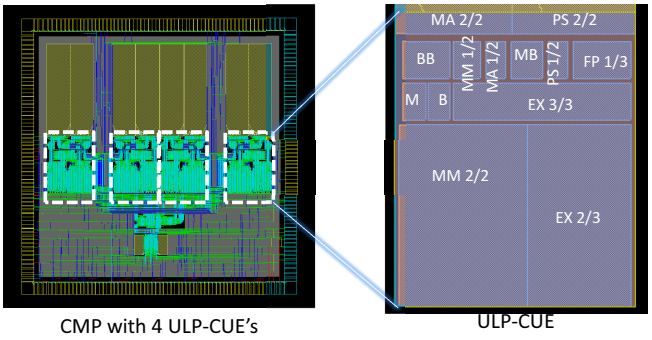


Fig. 7: Layout of prototype LSI.

and processing time of both the BMerge and BBranch is subtracted from the energy and execution time of the UDP/IP on the prototype LSI without any drawbacks on the CUE processors.

4.2 Power-performance verification

To verify the essential power consumption of the ULP-CUE, the energy required to execute the UDP/IP on the ULP-CUE is measured by using the prototype LSI. The program of the UDP/IP handling is consisted of two independent parts, sending-side and receiving-side programs. Figure 8(a) shows the measured energy for the sending-side and receiving-side programs. On the other hand, the total amount of executed instructions is shown for each program in figure 8(b). The energy increases approximately in proportion to the amount of executed instructions. This strong correlation is an evidence of the essential power consumption, i.e., the power is consumed only for instruction execution.

Additionally, the energy and execution time of the UDP/IP handling on the ULP-CUE is compared that of the UDP/IP handling on the CUE processors to show that the essential power consumption results in the energy reduction or ultra-low-power consumption. As discussed at the front of this section, the reduction of the execution time results in the energy reduction because the slack time can be used to lower the supply voltage. To show the total energy reduction, the energy required to execute the UDP/IP is estimated and then the energy reduction with low-voltage supply is estimated.

To estimate the energy of the UDP/IP handling on the CUE processor, the energy is measured by using the prototype LSI in which the bypass of the ULP-CUE is disabled and then the energy consumed by the BMerge and BBranch is subtracted from the measured energy. Generally, SPICE simulators are used to estimate energy of circuit but their transistor-level simulation requires tremendous amount of time to simulate the whole circuit of the practical processors with specific programs, and thus they are impractical in this estimation. In the circular pipeline, the energy is consumed mainly in the data-latch and logic circuit in every stage. The

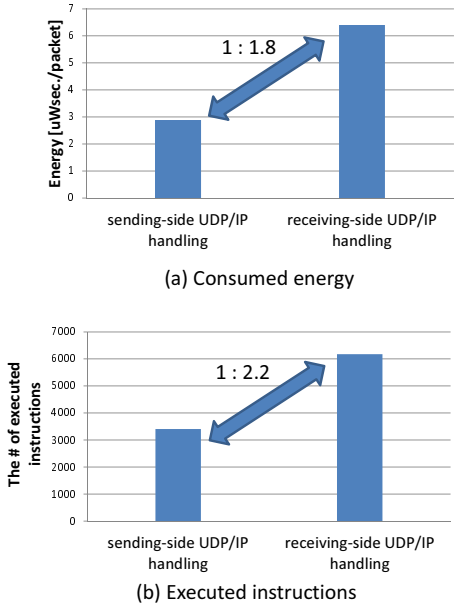


Fig. 8: An evidence of essential power consumption.

logic circuit of the BMerge and BBranch is consisted of only a MUX, and its circuit size is equal or smaller compared to the logic circuit of the other stages; therefore it has less energy consumption compared to the others. To estimate the energy consumed in the CUE processors without drawbacks on the CUE processors, the logic circuits are ignored and the energy consumed in the data latch is simulated by using the SPICE simulator. Table 9 shows the ratio of the simulated energy of the data-latch in every stage, and it reveals that the energy consumed in the BMerge and BBranch is 11.78% of the energy of the ULP-CUE. That is, the 11.78% of the energy measured with the bypass disabled should be subtracted to estimate the energy of the UDP/IP handling on the CUE processors. The result of the subtraction is shown in the figure 10.

As for the execution time of the UDP/IP handling, the time required to execute the instructions in the critical path of the UDP/IP handling is measured by using the prototype LSI and the measured results are shown in figure 11. Similarly to the energy estimated, the execution time of the UDP/IP handling on the CUE processors is estimated by subtracting the processing time of the BMerge and BBranch stages from the measured execution time. The processing time of the BMerge and BBranch is revealed by conducting gate-level simulation with the gate-net-list and delay information extracted from the layout of the ULP-CUE. The ratio of the revealed processing time of every stage is shown in table 9 in which it is shown that the processing time of the bypass is 6.89% of that of the circular pipeline of the ULP-CUE. Figure 11 shows the execution time estimated by subtracting the 6.89% execution time from the execution time of the UDP/IP handling on the ULP-CUE. Generally, the execution time of

Fig. 9: Estimated energy and processing time of each stage.

	Energy (ratio)	Processing time (ratio)
Merge	3.99%	18.59%
MM1/2	6.20%	9.20%
MM2/2	4.75%	5.92%
BMerge	3.93%	3.54%
PS1/3	4.45%	3.25%
PS2/3	3.77%	10.94%
PS3/3	8.47%	11.75%
FP1/2	10.20%	9.70%
FP2/2	31.95%	5.92%
MA1/2	4.65%	3.29%
MA2/2	4.08%	10.85%
BBranch	7.85%	3.35%
Branch	5.72%	3.70%

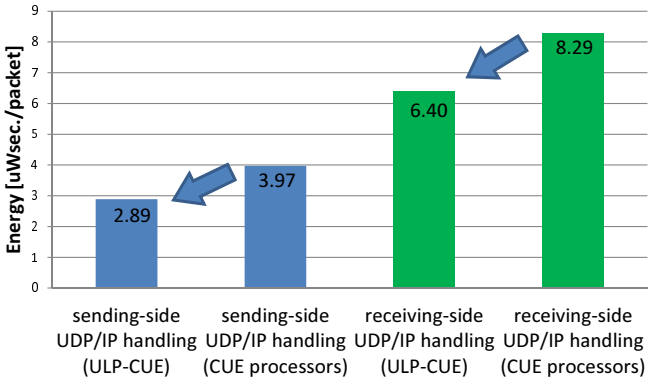


Fig. 10: Reduced energy.

programs is assumed to be in proportion to the supplied voltage, which is denoted by V_{DD}, and the energy required to execute the program is assumed to be in proportion to $(V_{DD})^2$. Based on these assumptions, the energy of the UDP/IP handling on the ULP-CUE is estimated as follows: $2.89 \times (\frac{344}{466})^2$ approximately equals 1.57 for the sending-side, and $6.40 \times (\frac{34,590}{47,581})^2$ approximately equals 3.38 for the receiving-side; therefor, $\frac{1.57+3.38}{3.97+8.29}$ approximately equals 40% for the UDP/IP handling.

Consequently, the essential power consumption of the ULP-CUE reduces the energy required to execute the UDP/IP handling to approximately 40% compared to the CUE processors.

5. Conclusion

In this paper, an optimized circular pipeline is revealed to realize real-time multiprocessing with essential power consumption toward ultra-low-power protocol handling, and its essential power consumption and power-performance is evaluated by using the prototype LSI of a data-driven

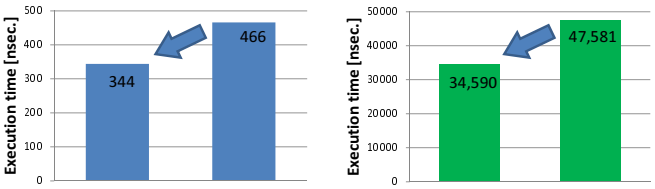


Fig. 11: Reduced execution time.

processor, which is named ULP-CUE and realized by the optimized circular pipeline. As a result of the evaluation, it is revealed that the ULP-CUE can execute the UDP/IP handling indispensable for ad hoc networking by consuming 40% of energy in comparison with ever-proposed implementations of the real-time multiprocessing.

The ULP-CUE is the central core to realize the networking platform in the ULP-DDNS project, and its circuit level power control scheme is also proposed in the project. The total energy of the networking platform realized by both the ULP-CUE and the power control scheme will be evaluated in our future works.

Acknowledgement

Although it is impossible to give credit individually to all those who organized and supported our project, the authors would like to express their sincere appreciation to all the colleagues in the project.

This research work was supported in part by Core Research for Evolutional Science and Technology (CREST), Japan Science and Technology Agency (JST). The circuit design work was supported by VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc. and Cadence Design Systems, Inc.

References

[1] "Mobile ad hoc networks (manet)." [Online]. Available: http://datatracker.ietf.org/wg/manet/charter/

[2] H. Nishikawa and S. Miyata, "Design philosophy of super-integrated data-driven processors: Cue," in *PDPTA*. WorldComp, July 1998, pp. 415–422.

[3] H. Nishikawa, K. Aoki, and H. Ishii, "Data-driven implementation of efficient protocol handlers," in *Proceedings of Scuola Superiore Guglielmo Reiss Romoli 2002w Computer & Internet Conference*, Jan. 2002, pp. (CD–ROM).

[4] H. Nishikawa, "Design philosophy of a networking- oriented data-driven processor-CUE," *IEICE Trans. Electron.*, vol. E89-C, no. 3, pp. 221–229, March 2006.

[5] H. Nishikawa, K. Aoki, H. Ishii, and M. Iwata, "Intermediate achievement of ultra-low-power data-driven networking system: ULP-DDNS," in *PDPTA*. WorldComp, July 2011, pp. 421–427.

[6] K. Miyagi, S. Sannomiya, M. Iwata, and H. Nishikawa, "Self-timed power-aware pipeline chip and its evaluation," in *PDPTA*. WorldComp, July 2011, pp. 442–448.

[7] S. Sannomiya, R. Kuroda, K. Aoki, K. Miyagi, M. Iwata, and H. Nishikawa, "Chip multiprocessor platform for ultra-low-power data-driven networking system: ULP-DDNS," in *PDPTA*. WorldComp, July 2011, pp. 428–434.

[8] C. J. Myers, *Asynchronous circuit design*. Univ. of Utah John Wiley & Sons, Inc., 2001.

Low-Powered Self-Timed Pipeline with Runtime Fine-Grain Power Supply

Kei MIYAGI[1], **Shuji SANNOMIYA**[2], **Makoto IWATA**[1], **and Hiroaki NISHIKAWA**[2]
[1]School of Information, Kochi University of Technology, Kami, Kochi, Japan
[2]Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba Science City, Ibaraki, Japan

Abstract— *This paper describes a runtime fine-grain power supply scheme based on the self-timed pipeline (STP) circuits. The STP works with its local hand-shake signal so that it does not require the global clock distribution, i.e., centralized control. Therefore, various power supply control for the STP can be naturally localized in both spatial and temporal domains without stopping its effective data transfer, e.g., program execution in case of microprocessors. As a result, the power supply scheme proposed in the paper can efficiently incorporate both commonly used voltage scaling and power gating techniques and it can further produce synergetic effects on its total power saving nature.*

In the paper, effective power-performance characteristics of the STP supported by the proposed scheme are discussed and analyzed in terms of break-even model for power reduction effect against its control overhead. Furthermore, the scheme is applied to an ultra-low-power data-driven networking processor, named ULP-CUE, designed in 65 nm CMOS process and then it is evaluated through typical UDP/IP traffic of a wireless ad hoc network. In this case, total power can be reduced to about 13% compared with the normal-STP-based data-driven processor.

Keywords: self-timed pipeline, runtime power gating, runtime voltage scaling, file-grain power-supply

1. Introduction

Lowering power dissipation of LSI systems is now more and more crucial to realize greener devices, while fully utilizing the potential speed of transistors under deep-submicron process technologies. In order to facilitate such low power consumption of the LSI chips, both dynamic and static power dissipation should be cut or reduced as much as possible. The main causes of the dynamic power dissipation are the transistor-switching unnecessary for processing and the excessive switching frequency higher than required processing speed, while the leakage current through inactive transistors increases the static power dissipation mainly.

Voltage scaling and power gating are well-known techniques for reducing switching power and static power respectively[1][2]. Mostly those techniques have been applied to relatively coarse grain power domain of the LSI chip, e.g., a whole die or processor core, and they are assumed to be controlled by the operating system. In order to achieve ultimate reduction of the power consumption, it is essential to control power supply more finely in terms of both spatial and temporal domains. To realize such fine control, a powered circuit block itself has to behave autonomously without any centralized control such as a global clock signal. Therefore, in our collaborative research project to establish ultra-low-power data-driven networking system (ULP-DDNS)[3], we have fully introduced self-timed pipeline (STP) circuits to implement our ultra-low-power data-driven processor (ULP-CUE) without any global clock signal. To minimize overheads caused by the fine-grain power control, our ultra-low-power self-timed pipeline (ULP-STP) circuit has been designed to work in part even during supply voltage scaling or power gating. That is, the ULP-CUE processor core can execute programs without evacuation of their contexts even when the power supply voltage is altered or power supply is cut in part.

Because of self-timed elastic data-transfer mechanism of the original STP [4], it can work well under variable voltage without adjusting clock frequency even if the altered voltage could transiently fluctuate at individual pipeline stage. Since the pipeline throughput can be adaptive to its processing load only by altering supply-voltage appropriately, a power-aware pipeline scheme can be realized naturally in terms of runtime power saving. For instance, the proportional-integral-derivative (PID) controller can be applied to such voltage control by monitoring consumption current of a target power domain within the chip.

The STP is also suitable for gating power-supply to fine grain circuits since its stage-by-stage data-transfer control independently activates only pipeline stages with valid data. We therefore proposed a stage-by-stage power gating scheme adopted in the STP [5]. This scheme provides natural signal gating, i.e., it stops the unnecessary signal propagation and transistor-switching at pipeline stage level without any global control mechanisms resulting in both power dissipation and processing speed degradation. Moreover, it makes it possible to scale the voltage even when the stages are activated because it can be realized without any global oscillator such as phase-locked loop (PLL) circuit, which forces pipeline flush ahead of the frequency and voltage change.

By introducing both the voltage scaling and power gating

into the STP, synergetic effects on its total power saving nature will be further expected. Apparently power gating will be conducted at the lowest voltage scaled so that the power overhead of power gating will be minimized.

As for the remaining part of this paper, the following section describes the circuit design of the ULP-STP with runtime fine-grain power-supply. Section 3 discusses a break-even model for power reduction effect against its control overhead to implement various ULP-STP systems. Section 4 shows the quantitative analysis of the ULP-CUE implemented by the optimized ULP-STP under 65 nm CMOS process and then we conclude in the final section.

2. Runtime fine-grain power-supply scheme for STP

By the runtime fine-grain power-supply scheme proposed in the paper, voltage scaling and power gating will be adaptively performed along with the present processing load within a system. In general, the processing load in the system may alter due to both the intrinsic parallelism of a program and the extrinsic request traffic to the program.

In this section, the autonomous behavior of the self-timed elastic pipeline (STP) is briefly introduced and then its natural contribution to the runtime fine-grain power-supply control is discussed.

2.1 Self-timed elastic pipeline

Each pipeline stage of the STP consists of a data latch as a pipeline register, function logic, and transfer control unit named C-element. The basic structure of the STP is shown in figure 1. The data latch, function logic, and C-element are denoted by DL, Logic, and C, respectively. The data is packed with tag into packet form, and the packet is transferred between the pipeline stages as a result of the communication between the C's in the adjacent stages. The communication is performed stage-by-stage according to the 4-phase handshake protocol [6] by using transfer request and acknowledge signals which are called send signal and ack signal respectively. The stage-by-stage transfer control changes the states of each pipeline stage independently, and the states of the stages are defined below according to the handshake protocol. Here, the C-element in the i-th stage is denoted by C_i.

- Reset state: The send and ack signals are negated after the assertion of the reset signal.
- Idle state: The C_i waits until the $send_{i-1}$ is asserted.
- Busy state: The $send_{i-1}$ is asserted at the beginning of the transfer of the packet from the precedent $(i-1)$-th stage. After the assertion of the $send_{i-1}$, the C_i asserts its ack signal (ack_{i-1}). In response to the assertion, the C_{i-1} negates the $send_{i-1}$. After that, if and only when both the $send_{i-1}$ and ack_i are negated, the C_i asserts the $ToDL_i$ to open the DL_i and it asserts $send_i$ at the same time. As a consequence, the packet is latched in the i-th stage, and the i-th stage goes to idle state. Otherwise, the C_i waits until the ack_i is negated while it keeps its send and ack signals.

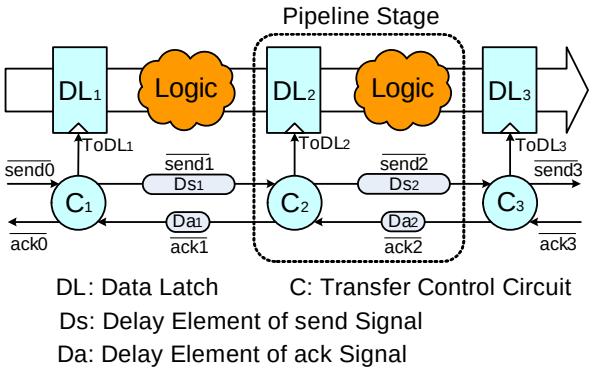


Fig. 1: Basic structure of self-timed pipeline.

The successive stages receiving the assertion of the send signal go to busy state and their C's repeat the same transfer control sequence individually. During the handshakes, the send signals are delayed to assure the completion of the primitive logic function and ack signals are delayed to assure the setup-hold timing of the DL's.

This stage-by-stage transfer control of the STP suggests the timing of the power controls. That is, in the idle stages, the circuit of the DL, and combinational Logic can be powered-off, i.e., the supply-voltage can be cut while that of the C and sequential Logic can be powered-down, i.e., the supply voltage can be lowered enough to keep the circuit's states. Moreover, in the busy stages, those circuits should be powered-down enough to assure the switching of the transistors, i.e., the supply-voltage can be lowered as long as the required switching speed is achieved.

2.2 Stage-by-stage power gating

To realize stage-by-stage power controls finely, we have already proposed an ultra-low-power self-timed pipeline (ULP-STP) structure illustrated in figure 2 [5]. In the ULP-STP structure, V_{DD} supplied to all circuits is scaled by using DVS technique. In addition, to cut the power-supply to the DL and Logic, a high threshold NMOS transistor, called power switch (PS), is placed between V_{SS} and the ground-side terminals of the DL's and combinational logics which are composed of low-threshold transistors. In this case, an isolation element (ISO) must be inserted between adjacent stages in order to block the propagation of the electrically unstable signals from the gated stages to the other active stages. This ISO function can be implemented in a part of a data-latch so that the circuit overhead (i.e., power and delay time) for the ISO is negligible. Each power switch is controlled by its power control circuit (PC) which observes the send and ack signals.

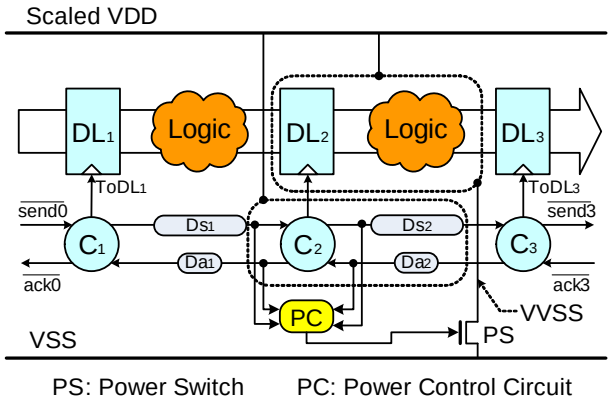


Fig. 2: Power-supply control for STP.

The ULP-STP structure makes it possible to power-down or power-off stage-by-stage, and thus the necessary power is supplied only to the processing stages having a valid data while the leakage power at other idle stages are finely cut. Therefore this power gating scheme is named runtime fine-grain power gating (RTPG) in this paper. In addition, the power gating for each core is achieved without any additional mechanism because all of the stages in an idle core are powered down as a result of the stage-by-stage power gating in each stage. Since each stage will consecutively wake up along with travelling of a newly arriving data within the STP, total rush current of the whole STP system can be temporally distributed when the PS is powered on.

2.3 Runtime voltage scaling

With the proposed structure, the DVS and the PG can be enabled independently or simultaneously. The supply voltage to the STP can be autonomously altered without adjusting the clock frequency since the STP itself is clockless. The supply voltage can be controlled based on the consumption current which is proportional to the amount of processing load in the STP system.

On scaling the supply voltage in runtime, sharp scaling may cause overshooting or undershooting the voltage and thus power noise and malfunction may be generated. On the other hand, dull scaling may bring down to degrade its prompt operation. Since the consumption current of the STP may fluctuate usually, adaptive control mechanism is necessary. Thus, we introduce the proportional-integral-derivative (PID) controller to our voltage scaling scheme to minimize the error by adjusting the process control inputs, i.e., consumption current values.

Figure 3 shows the runtime voltage scaling mechanism (RTVS) introducing the PID controller. At the I-V lookup table in the figure, the appropriate supply voltage is indexed by a sampled consumption current value. For instance, the target voltage in the I-V table may be designed to achieve the maximum throughput per power. How to optimize the

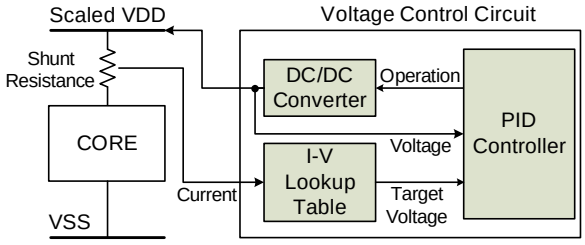


Fig. 3: Voltage controller.

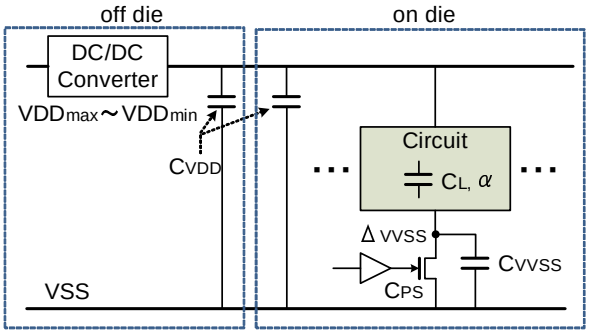


Fig. 4: Equivalent load capacitance of ULP-STP.

I-V table may depend on the low-power policy of the target application. Note that a transfer function of a target system can be simply approximated by its dead time and first-order lag. Thus, the PID parameters for our RTVS can be adjusted by using the same approximation.

Those advanced features indicate that the STP is more robust than the clocked pipeline, especially in the nanometer-scale processes with more variation in transistor performance.

3. Break-even model of the ULP-STP

Although voltage scaling and power gating can reduce the switching power and the leakage power of the transistors respectively, there are power overheads such as charge/discharge current during scaling supply voltage and switching current of the power switch transistors for the power gating, and so on. In this section, a break-even model of power reduction effect against its overhead is discussed, especially in terms of runtime and fin-grain of RTPG and RTVS.

Figure 4 illustrates a simple equivalent circuit model of the ULP-STP. With the voltage scaling, there are equivalent load capacitances C_{VDD} of on-die/off-die power lines and. With the power gating, there are some equivalent load capacitances, C_L, C_{VVSS}, and C_{PS} each of which is extracted from the target power domain circuit, the virtual ground line (VVSS), and the power switch (PS) respectively. Here, the switching energy of the PS, E_{PS}, is represented as follows:

$$E_{PS} = C_{PS} \times VDD^2 \quad (1)$$

It is assumed here that the power of ISO is negligible because the ISO can be overlapped onto the data-latch. When the power switch is powered on, the power consumption due to the rush current to the circuit, E_{rush}, can be calculated based on [7] as follows:

$$E_{rush} = \left(C_{VVSS} + \frac{1}{2} C_L \right) \times VDD \times \Delta_{VVSS} \quad (2)$$

where Δ_{VVSS} denotes the increased amount of VVSS before wake-up. Based on equations 1 and 2, the lower bound of the sleep time to get power reduction effects, BET can be represented as follows:

$$BET = \frac{C_{PS} \times VDD^2 + \left(C_{VVSS} + \frac{1}{2} C_L \right) \times VDD \times \Delta_{VVSS}}{P_{leak}} \quad (3)$$

where P_{leak} denotes the leakage power of the target power domain circuits.

As for the break-even condition related the voltage scaling, it depends on the processing load. Thus, the break-even processing load, $BEPL$ can be expressed as follows:

$$BEPL = \frac{C_{VDD}}{C_L \times \alpha} \quad (4)$$

where α denotes the average switching provability of the transistors within the target power domain circuits. Although the PID controller itself must consume the energy, it can be negligible compared to the charge/discharge energy of C_{VDD}.

The equation 3 and 4 represents the basic break-even conditions of the proposed runtime fine-grain power-supply control scheme. As represented in those equations, the break-even conditions depend on those load capacitances of the target system. In the following section, the low-power characteristics of the ULP-CUE as an actual STP system will be evaluated through the execution of a practical networking application program.

4. Power-performance evaluation

In our collaborative research project on ultra-low-power data-driven networking system (ULP-DDNS), a data-driven processor ULP-CUE based on the self-timed pipeline has been implemented by using 65 nm CMOS process. In this section, the ULP-CUE is briefly introduced and then the basic power-performance characteristics are evaluated by integrating actual measurement results of the ULP-CUE chip and SPICE simulation results. Finally, total power reduction effects of the proposed runtime fine-grain power-supply are revealed in the case the ULP-CUE executes a UDP/IP protocol handling program based on a typical traffic log of an ad hoc wireless network simulated by the network simulator[8].

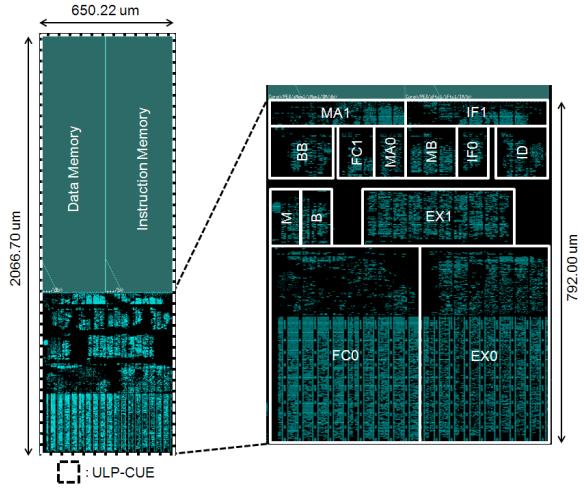


Fig. 5: Layout of the experimental ULP-CUE (65nm CMOS 7ML process).

4.1 Circuit configuration of ULP-CUE

The ULP-CUE is a 32 bit dynamic data-driven processor implemented by the 13-stages ring-shaped STP and each STP stage is composed of the following elemental functional module.

- M: merging function of input tokens and internally circulated tokens.
- FC: firing control function to detect a pair of operand tokens for its instruction execution. It is divided into two STP stages, FC0 and FC1.
- MB: merging function for tokens bypassing the FC stages.
- IF: instruction fetching function. It is divided into two stages, IF0 and IF1.
- ID: instruction decoding function.
- EX: execution function, i.e., ALU. It is divided into two stages, EX0 and EX1.
- MA: data-memory access function. It is divided into two stages, MA0 and MA1.
- BB: branch function to bypass the FC stages or not.
- B: branch function to ether output port or the circular STP.

Those stages are placed and routed on a die shown in figure 5. As shown in the figure, area of each stage is different from others so that the load capacitance of each stage is different. This means its break-even condition is different.

4.2 Evaluation procedure

Because each STP stage of the ULP-CUE is implemented as different circuits, the break-even time is different. For each stage, it is difficult to measure every parameter in equation 3. Thus, in this evaluation, PS switching energy E_{PS}, energy consumption caused by rush current E_{rush},

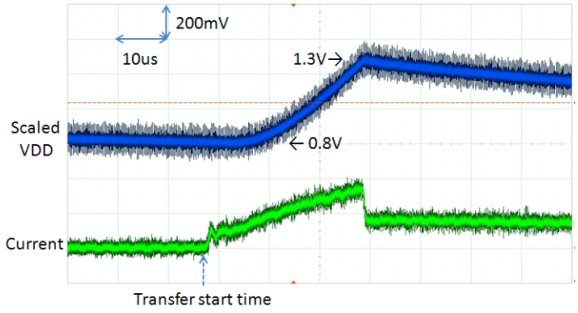


Fig. 6: An example measurement result of RTVS (25°C).

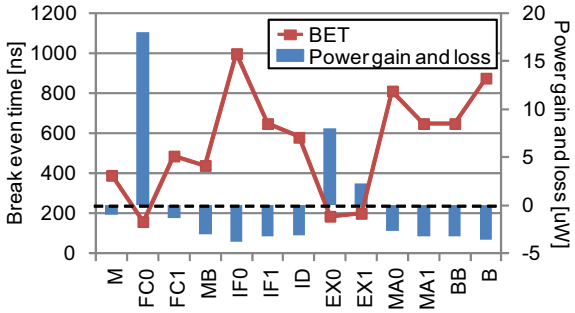


Fig. 7: Break even time of each STP stage (0.8 V, 25°C).

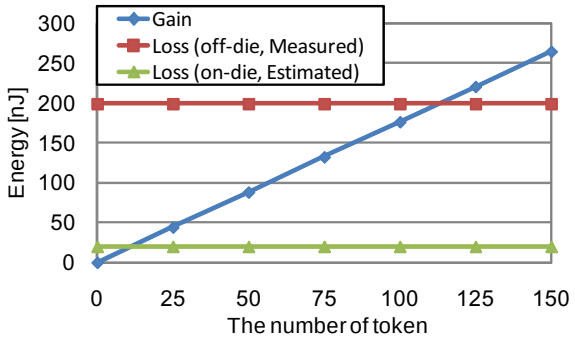


Fig. 8: Break even processing load of ULP-CUE (0.8 V 1.2 V, 25°C).

and leakage power P_{leak} are evaluated by SPICE simulation of each stage. This is because the detailed breakdown of each stage's power consumption cannot be measured on the fabricated ULP-CUE chip. Since the voltage of the VVSS depends on sleep time, the SPICE simulation is conducted in many times in the case of different sleep time. Furthermore, the gate width of the power switch NMOS transistor is designed to reduce the voltage drop to less than 5%. Wake-up delay time of the stage is hidden by asserting a power-on signal from the preceded STP stage.

As for the voltage scaling, total power of the ULP-CUE processor can be measured on the fabricated chip as shown in figure 6. This measured wave shows an example of consumption current of the ULP-CUE in the case the supply voltage VDD is changed from 0.8 V to 1.3 V by using the PID controller. This consumption current includes charge current to both C_{VDD} and C_L. Since the VDD can be forced to alter from 0.8 V to 1.3 V, only the charge current to C_{VDD} can be measured. Thus, the difference of both measurement values expresses that to C_L. As a result, the break-even processing load can be calculated based on the equation 4.

4.3 Basic evaluations on break-even model

As described in the previous subsection, the basic characteristics of power consumption in the ULP-CUE have been evaluated in the case it execute the UDP/IP protocol handling program. Figure 7 shows the break-even time and power reduction effect of each STP stage composing the ULP-CUE at 0.8 V, 25°C. The solid line shows the break-even time, and the solid bar shows the amount of power reduction when the sleep time is 250 ns. If the sleep time is longer than 250 ns, the total power reduction of the ULP-CUE can be gained by the stage-by-stage power gating.

The FC0 stage has the shortest BET, 159 ns. This is because its area is the largest in all stages. Furthermore, the gate width of the PS can be shortened because the switching probability of transistors composing the FC0 is not so high compared with other stages. The IF0 has the longest BET, 839 ns. It is about 5 times longer than the shortest one. Those results implies that it is important to introduce an adaptive power gating mechanism such as an invalidation scheme of individual power gating based on a leakage current monitor[9].

As for the voltage scaling, the break-even processing load is evaluated based on the equation 4. Figure 8 shows the break-even processing load of the ULP-CUE based on the measurement current of the chip when the supply voltage is changed from 0.8 V to 1.2 V. The diamond-shape plots indicate C_L, i.e., the denominator part of the equation 4, and the square-shape plots indicate C_{VDD}, i.e., the numerator part of that. From this result, the BEPL is about 113 tokens.

In order to estimate an acceptable voltage scaling frequency, potential performance-power characteristics of the ULP-CUE have been measured at 0.8 V to 1.3 V. Those results are shown in figure 9. From this result, the maximum performance of the ULP-CUE is 87K UDP/IP packet/sec. at 1.3 V where the length of a UDP/IP packet is 512 bytes. When the throughput of UDP/IP packets is 29K packet/sec., total power consumption at 0.8 V can be reduced to 38% compared to that of 1.2 V. Therefore, the evaluated BEPL, i.e., 113 tokens, indicates that the acceptable voltage scaling frequency is at most 32.7 KHz.

The measured chip is not equipped with the on-die DC-DC convertor. If a DC-DC convertor can be implemented on a die, the load capacitance of the power line, C_{VDD},

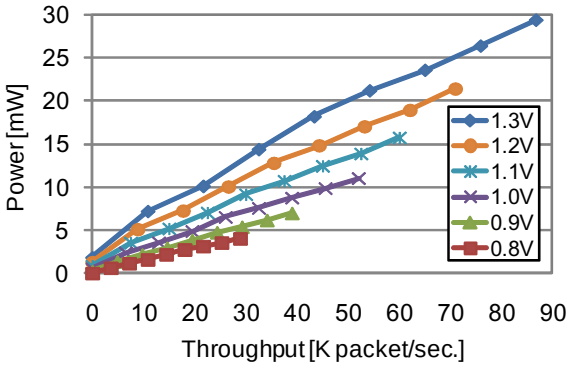


Fig. 9: Performance-power characteristics of the experimental ULP-CUE under altered supply-voltage (25°C).

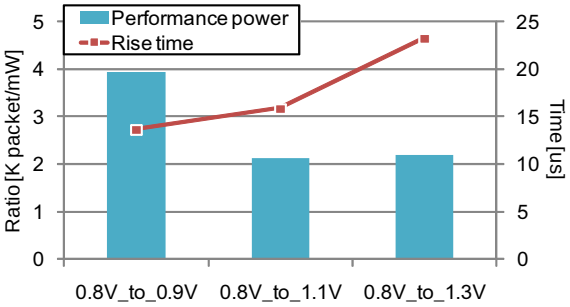


Fig. 10: Performance-power characteristics of RTVS (25°C).

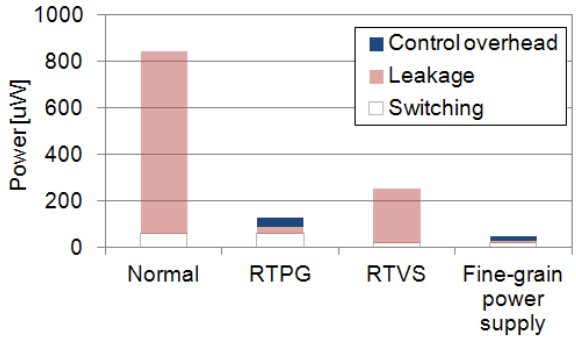


Fig. 11: Power comparisons of the ULP-CUE (54 M bit/sec., 25°C).

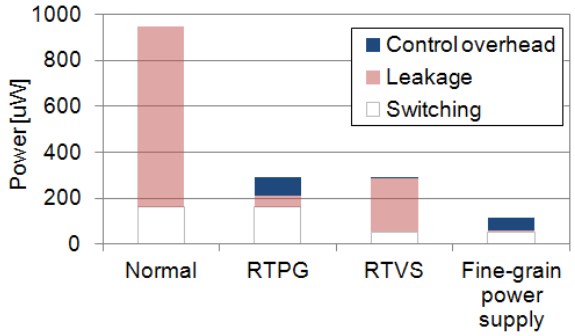


Fig. 12: Power comparisons of the ULP-CUE (162 M bit/sec., 25°C).

can be reduced to one-tenth of that. In this case, the break-even processing load can be reduced to 11 tokens. This assumption indicates that the runtime voltage scaling may be conducted at most 336 KHz.

Figure 10 shows the measured transient power-performance ratios and voltage rise times when the supply-voltage is altered from 0.8 V to 0.9 V, 1.1 V, and 1.3 V. Even during such transient time of supply-voltage, the ULP-CUE can work at reasonable power-performance ratio. Therefore, total performance-power ratio could be improved as well as better dependability against hard real-time constraints can be obtained.

4.4 Evaluations on typical network traffic

Since the ULP-CUE has been designed to implement network protocol handling efficiently, its practicality should be evaluated based on actual network traffic pattern. Therefore, we used a network simulator, OPNET, to obtain such benchmark traffic logs. In this evaluation, an ad hoc wireless network [8] is assumed and then simulated by OPNET to get traffic logs. From those traffic logs, a set of input token to the ULP-CUE is extracted and the data-driven UDP/IP program is executed on it.

In the case that the basic wireless transmission rate of this network is 54M bit/s or 162M bit/s, the power consumption of the ULP-CUE is estimated based on the above basic evaluation results. Figure 11 and 12 show the power comparison among the normal STP, the STP with RTPG, the STP with RTSV, and the STP with the proposed runtime fine-grain power-supply scheme (i.e., with both RTPG and RTSV).

In figure 12, the STP with RTPG can reduce its leakage power to 48uW. This is 6% leakage power compared to the normal STP. The STP with RTVS can reduce its switching power to 51.7uW and its leakage power to 235uW. This is 32% switching power compared to the normal STP. In the case of the STP with the proposed power-supply scheme, the ULP-CUE can reduce the total power to 13%. Because of the synergetic effects between RTPG and RTVS, the switching power of the PS in RTPG can be reduced to 68%.

5. Conclusion

In this paper, a runtime fine-grain power-supply mechanism based on the self-timed elastic pipeline (STP) was proposed to realize lower-power LSI circuits and then it was analyzed by defining a break-even model in terms of power trade-off. The proposed circuit introduced both the power gating and voltage scaling techniques so that it could utilize

the synergetic effects between them. The low-powered STP circuit was then applied to an ultra-low-power data-driven processor, ULP-CUE, and evaluated through typical UDP/IP traffic of a wireless ad hoc network. In this case, total power can be reduced to about 13% compared with the original STP-based CMP.

Since the break-even condition of the proposed scheme may change depending on the temperature and process variations, a kind of self-checking circuit of typical leakage and switching power should be introduced on a die and its monitoring result should be dealt with as a feedback to power-supply controller. Furthermore, in order to verify such on-die mechanism in terms of power consumption and performance, a microarchitecture simulator must be developed which can simulate not only architectural behavior but also transient power consumption. We are now developing such a platform simulator [10] in our collaborative research project and then we will report the comprehensive evaluation results using this simulator iin near future..

Acknowledgement

Although it is impossible to give credit individually to all those who organized and supported our project, the authors would like to express their sincere appreciation to all the colleagues in the project.

This research work was supported in part by Core Research for Evolutional Science and Technology (CREST), Japan Science and Technology Agency (JST). The circuit design work was supported by VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc. and Cadence Design Systems, Inc.

References

[1] P. Pillai and K. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," Proc. of The 18th ACM Symposium on Operating Systems Principles (SOSP'01), pp.89–102, October 2001.

[2] S. Mutoh, S. Shigematsu, Y. Gotoh, and S. Konaka, "Design method of MTCMOS power switch for low-voltage high-speed LSIs," Proc. of Asia and South Pacific Design Automation Conference(ASP-DAC'99), pp.113–116, Jan. 1999.

[3] H. Nishikawa, K. Aoki, H. Ishii and M. Iwata, "Intermediate Achievement of Ultra-Low-Power Data-Driven Networking System: ULP-DDNS," Proc. of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications(PDPTA'11), pp.421–427 July 2011.

[4] H. Terada, S. Miyata, and M. Iwata, "DDMP's: Self-Timed Super-Pipelined Data-Driven Processors," Proc. of the IEEE, Vol.87, No.2, pp.282–296, February 1999.

[5] S. Sannomiya, K. Miyagi, K. Sakai, M. Iwata, and H. Nishikawa, "Self-timed power gating for ultra-low-power pipeline circuit," Proc. of the 2009 International Conference on Parallel and Distributed Processing Techniques and Applications(PDPTA'09), pp.575–580, July 2009.

[6] C. J. Myers, "Asynchronous circuit design," Univ. of Utah John Wiley & Sons, Inc., July 2001.

[7] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, "Microarchitectural Techniques for Power Gating of Execution Units," ISLPED, pp.32–37, Proc. of the 2004 International Symposium on Low Power Electronics and Design (ISLPED'04), August 2004.

[8] K. Utsu, H. Sano, C. Chow, and H. Ishii, "Proposal of Load-aware Dynamic Flooding over Ad Hoc Networks," Proc. of IEEE TENCON 2009, THU2, pp.1–6, November 2009.

[9] N. Seki, L. Zhao, J. Kei, D. Ikebuchi, Y. Kojima, Y. Hasegawa, H. Amano, T. Kashima, S. Takeda, T. Shirai, M. Nakata, K. Usami, T. Sunata, J. Kanai, M. Namiki, M. Kondo and H. Nakamura, "A Fine Grain Dynamic Sleep Control Scheme in MIPS R3000" the 26 IEEE International Conference on Computer Design (ICCD'08), pp.612–617, October 2008.

[10] K. Aoki, H. Ishii, M. Iwata, and H. Nishikawa, "A Comprehensive Evaluation of ULP-DDNS by Platform Simulator," Proc. of the 2012 International Conference on Parallel and Distributed Processing Techniques and Applications(PDPTA'12), PDP6025, July 2012 (to be presented).

A study on Overload-Avoidance Scheme of ULP-DDNS for Congestion-Free Networking System

Yukikuni Nishida and Hiroaki Nishikawa
Graduate School of Systems and Information Engineering, University of Tsukuba
Tsukuba Science City, Ibaraki 305-8573 Japan

Abstract – ***Since an ad hoc network does not need a base station and can constitute a network locally, it is important for a sensor network or a temporary network in emergency. Especially at the time of a disaster, power-saving is required for a long battery life. Since the power consumption is proportional to a processing load linearly and the essential power is only consumed, ultra-low power data-driven networking system (ULP-DDNS) utilizing a passive data-driven principle is low power consumption. We consider that the feature that processing time does not change in multi-processing enables an overload condition to be avoided in advance if a content of a process is obtained. In this paper, we describe an overload-avoidance scheme for congestion-free networking system.***

Keywords: ad hoc network, Data-driven networking system, Congestion-free, Overload avoidance

1 Introduction

Wireless ad hoc network communication does not need an infrastructure such as an access point or a base station, and transmits information by a function being had in a node and hop-by-hop manner. Therefore, it attracts attention as an urgent communication tool when an infrastructure is down in a time of disaster. However, in order that a host may move, communication route changes. It is difficult to determine the communication route in that situation. A simple flooding is used as a one method to solve the problem. The simple flooding makes an unwanted traffic increase in ad hoc network, it causes increases of collision and retransmit, and it causes a congestion finally.

In order to reduce the increases in the unwanted traffic, efficient information discovery system [1] and efficient broadcast type transmission mode [2] have been proposed. Moreover, in an ad hoc network, since the certificate authority in an infrastructure cannot be used, a communication partner cannot check whether you are a partner who wants truly. Therefore, the efficient technique of attesting a communication partner from the confidential relation between nodes has been also proposed[3].

Furthermore, energy saving is required for ad hoc network node which is supplied energy by a battery in many cases in order to enable a long time communication. Especially in case of disaster, electric power may be lost in a long time, thus more energy saving is required to the network nodes. In order to satisfy these requirements, we paid our attention to a feature of a data-driven processor of which processing period is constant even for multi-processing condition required by a network processing and a self-timed elastic pipeline which consumes energy only in pipeline stages in which valid data exists. And, we have developed a data-driven processor with a self-timed elastic pipeline as hardware realization of the processor [4][5].

In order to achieve congestion-free ad hoc network communication, it is also necessary to avoid an overload condition of network nodes for preventing degradation of the communication quality. There are two reasons that congestion occurs. One is that a traffic exceeding a bandwidth of a link between nodes flows into the link. Another is that a traffic exceeding a processing performance of a node flows into the node. Because the node can control the flow when a processing load is low, the congestion can be avoided. However, when the node is overload, the node cannot also control the flow. Therefore, an overload of the node must be avoided. A processing period on ULP-DDNS is predictable as described before, and if a traffic pattern is obvious, overload avoidance can be performed in advance. There, in this paper, we considered about an overload avoidance scheme being appropriate for ULP-DDNS applied in ad hoc network.

2 Ultra-low-power data-driven Networking System: ULP-DDNS

ULP-DDNS is a networking system which can provide longer lifetime or stronger dependability under severe power budget and traffic condition. Each networking node only consumes the minimum and essential power without any runtime overhead by ultimately utilizing passive data-driven principle throughout ad hoc networking scheme, CMP platform dedicated to multiple UDP/IP processing and its self-timed VLSI realization.

2.1 ULP-DDNS node

Fig. 1 shows a block diagram of ULP-DDNS node. This ULP-DDNS node is constructed so that ad hoc networking system is executed on an application processor and UDP/IP

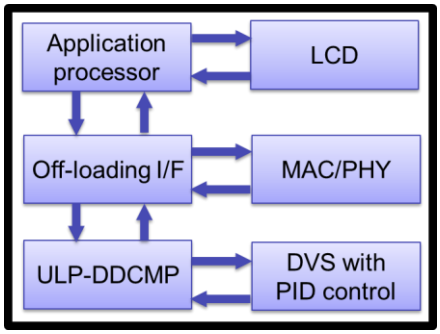


Fig. 1. Block diagram of ULP-DDNS

processing is executed by offloaded to ULP-DDCMP. Therefore, ULP-DDCMP and the application processer are embedded in ULP-DDNS. However, the architecture of ULP-DDNS is not restricted by this architecture. It is assumed that the ad hoc networking system is executed on ULP-DDCMP in future.

An electrical power for ULP-DDCMP is supplied by using dynamic voltage scaling (DVS) with PID control, a processing load of ULP-DDCMP is estimated by an observed electrical current consumption of ULP-DDCMP and electrical power is supplied by autonomously controlled voltage according to the processing load. In this way, DVS is constituted by a simple mechanism without observing an internal condition of ULP-DDCMP.

Off-loading I/F translates data received from MAC/PHY or the application processor into tokens by 4 Byte so that ULP-DDCMP is able to process them, and sends the tokens to ULP-DDCMP adequately. The I/F coordinates a timing between synchronous and asynchronous signals since ULP-DDCMP performs in asynchronous. In this paper, ULP-DDNS node is used to explain an operational timing of some kind of processes and packet sending and receiving.

2.2 Ultra-low-power data-driven chip multiprocessor: ULP-DDCMP

ULP-DDCMP is a chip multiprocessor in which four ULP-CUEs are connected by a token router. The tokens sent to ULP-DDCMP are sent to either ULP-CUE through the token router. Since an electrical current of each ULP-CUE is observed individually, a load distribution is also possible by observing electrical current varying according to processing load of each ULP-CUE.

2.3 Ultra-low-power data-driven CUE: ULP-CUE

ULP-CUE is constituted by a self-timed elastic pipeline of a circular type with short-cut path in order to shorten a turn-around-time of unary operations and to improve a processing performance. A feature of ULP-CUE is that a processing latency is constant at multi-processing condition

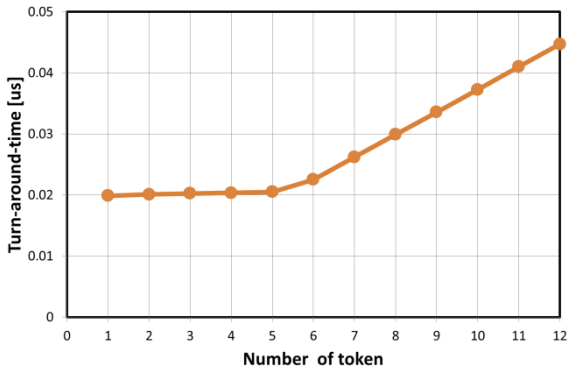


Fig. 2. Turn-around-time of ULP-CUE

unlike von Neumann processor. Since ULP-CUE is constituted by a circular pipeline, an execution time for one instruction is a turn-around-time that a token goes around the circular pipeline. The number of multi-processing is also represented by the number of tokens existing in the circular pipeline. Fig. 2 shows the experimental result of the turn-around-time of when the number of tokens in the circular pipeline is varied by repeated issuing multiply instructions. Fig. 2 shows that the turn-around-time is constant when the number of tokens is between one to five, therefore, a constant processing time is kept by five multiple process.

In this way, it is studied about overload avoidance scheme for congestion-free network of when ULP-DDNS which is appropriate for a network processing and is ultra-low power is applied to ad hoc network.

3 Urgent ad Nohoc networking system

3.1 Migration to an urgent ad hoc network

Network nodes communicating through base stations of an infrastructure when the infrastructure can be used will migrate to an ad hoc network and will try to communicate with the other nodes if the communication between the nodes and the base stations is lost by disaster etc. the nodes migrating ad hoc mode send packets to neighbor nodes by using flooding, and the neighbor nodes relay the packets as the relay nodes. The packets spread in whole ad hoc network, it increases traffic. Since the location of the relay nodes is determined in ad hoc network, traffic concentrates at a part of the relay nodes. Therefore, a collision in wireless area and traffic exceeding a processing performance of the nodes cause congestion.

To achieve an efficient ad hoc communication by solving these problems, some schemes are proposed [1][2][3]. It is explained that a discovery of a node stored any information, an authentication of the node, and obtaining data from the node which are assumed to be used in urgent ad hoc network communication

3.2 Discovery

There is a demand that it wants to know a current state of a particular place at the time of disaster etc. A method finding a network node being in this particular place and having desired information has been proposed, and the method makes an inquiry efficient. A procedure is as follows.

- A node requiring to obtain an information sends a query packet. A reply area is determined an angle a on either side of the line that connects itself with the destination.
- A node in a reply area calculates a distance of a destination node coordinate included in the query packet and a local node coordinate obtained by GPS and reply.
- The node requiring the information requests the discovery to a node which is closest to the destination
- By a repeat of these operations, a node which is closest to the destination is selected, and offers information.

In this way, the node holding necessary information is discovered efficiently.

3.3 Authentication

It is necessary to confirm whether a communication partner is correct or incorrect even if the communication uses ad hoc network. A public key certificate collection method based on a trust relationship list has been proposed. The method collects only the necessary certificate when a communication is required. Some trust relationships are created between neighbor nodes in the initial phase of constructing the network. Based on the initial trust relationship, the nodes collect the information of the trust relationship by a trust chain. In this way, the method makes authenticating a node efficient

3.4 Forwarding

As described above, since it is difficult to determine a route in ad hoc network, a communication utilizes flooding mainly. However, simple flooding causes many collisions. Information can reach only nodes of 70% in ad hoc network by the collision. Therefore, Load-aware Dynamic Counter-based Flooding (LDCF) has been proposed. LDCF forwards information efficiently by using a load state of local node.

LDCF measures a current processing load of a local node by a queue length in layer 2. LDCF judges by a queue length of to whether send or discard packets received at node. When the queue length is large, the queue length makes LDCF conscious of the high-load of a node and LDCF inhibits the forwarding packets. An inhibition of a sending packet is operated as follows.

- One or more relay nodes transmit the same packets from a same information source and the number of times same packets reception caused by above reason is counted in certain interval.
- Thresholds which allow transmission dynamically are configured for each information source and each packet, and the transmissions are canceled if the number of counts exceeds the threshold.

In this way, packets are forwarded efficiently.

4 Overload avoidance for congestion free network

4.1 Condition for avoiding overload

Generally, there are two reasons that a network go into congestion condition. One is the case that a packet is not able to send by a collision caused by attempting a packet sending of a traffic exceeding the network bandwidth. Another is the case that packet is not able to forward by receiving packets exceeding a processing performance of a network node. Former is able to give a notice to reduce a transmission rate to a packet sending host by checking a status of a packet sending. However, since latter is in overload condition, it cannot give a notice to another host to reduce a transmission rate based on a status of a packet sending. Therefore, it is necessary to consider a scheme to avoid an overload before a node becomes overload.

When ULP-DDNS is used, a constancy of processing period is kept even if processes are multiple by the feature of ULP-DDCMP and ULP-CUE as described above. Here, IEEE802.11 standard which is typical wireless communication method is described. IEEE802.11 has adopted Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA) as an access control, and a data frame transmission is prohibited during a back-off control period even if wireless area is not used. 802.11 headers are also added to UDP/IP packet. These overhead decrease an actual transmission rate. For instance, in the case of IEEE802.11g, maximum wireless transmission rate is 54Mbps, however, UDP/IP transmission rate is less than 25Mbps. period when wireless area is used by sending and receiving packets depends on UDP/IP packet length. Therefore, minimum interval of sending and receiving packets depend on UDP/IP packet length. ULP-DDCMP will not be an overload if a packet process is completed in a period shorter than this minimum packet receiving interval as one view.

In this paper, as an example, Discovery (Phase 1), Authentication (Phase 2), and forwarding (Phase 3) which are needed to obtain certain information by using ad hoc network communication were considered about conditions for

overload avoidance for starters. As an ad hoc network used by this examination, we assumed that ten nodes are placed at same distance on a line. A wireless range of each node also assumes three nodes. Then, sending and receiving data frames, a receiving process (Rx), a forwarding process (Fwd), and a sending process (Tx) are considered in ULP-DDCMP at a request node, a relay node, and an information node. Moreover, UDP/IP packet lengths by using discovery, authentication, and forwarding are 100Byte, 1000Byte, and 500Byte are assumed, respectively. The packet lengths are average packet lengths in each protocol. Then, in this examination, LDCF sending packet efficiently was used since it decreases flooding in wireless range at transmission

4.2 Discovery phase

The state of packet sending and receiving and internal processing in node 1, 4, 10 is shown in Fig. 3. Node 1 requires to obtain a information, node 4 receives a request as next discovery node, and node 10 holds the required information. Moreover, Fig. 3 shows the process until an information discovery is completed. Colored rectangles represent periods that radio wave is transmitted and received in wireless area, Rx, Fwd, and Tx represent periods of a receiving process, a forwarding process, and transmitting process, respectively. Additionally, a downward arrow shows that MAC/PHY tries to transmit a packet, however, that the transmission is waited because the wireless area is used by receiving packet.

At first, node 1 sends a query packet by broadcast to discover node 10 having an information being desired by node 1. Node 2, 3, and 4 which receive the query packet from node 1 reply distances between a destination coordinate and each node according to the information discovery protocol. Node 1 sends a request packet to request next information discovery to node 4 which is nearest to node 10 in the received reply packet. Node 4 which received the request tries the information discovery similar to node 1. In Fig. 3, a processing flow of the discovery processing in node 4 is omitted because it is same as the processing flow in node 1. When the request packet is received at node 10 that is the information source, the destination coordinates are forwarded to node that transmitted this request packet. The packet replied from node 10 is sent to the node having requested the information discovery. Finally, this information discovery is completed when the reply packet arrives at node 1.

The examination that uses Fig. 3 leads following result.

- Since destination of packets received at when each node perform a transmission process is not local node, processes does not overlap in ULP-DDCMP
- If a period of packet receiving process is shorter than a packet receiving interval, processes will not overlap in ULP-DDCMP.

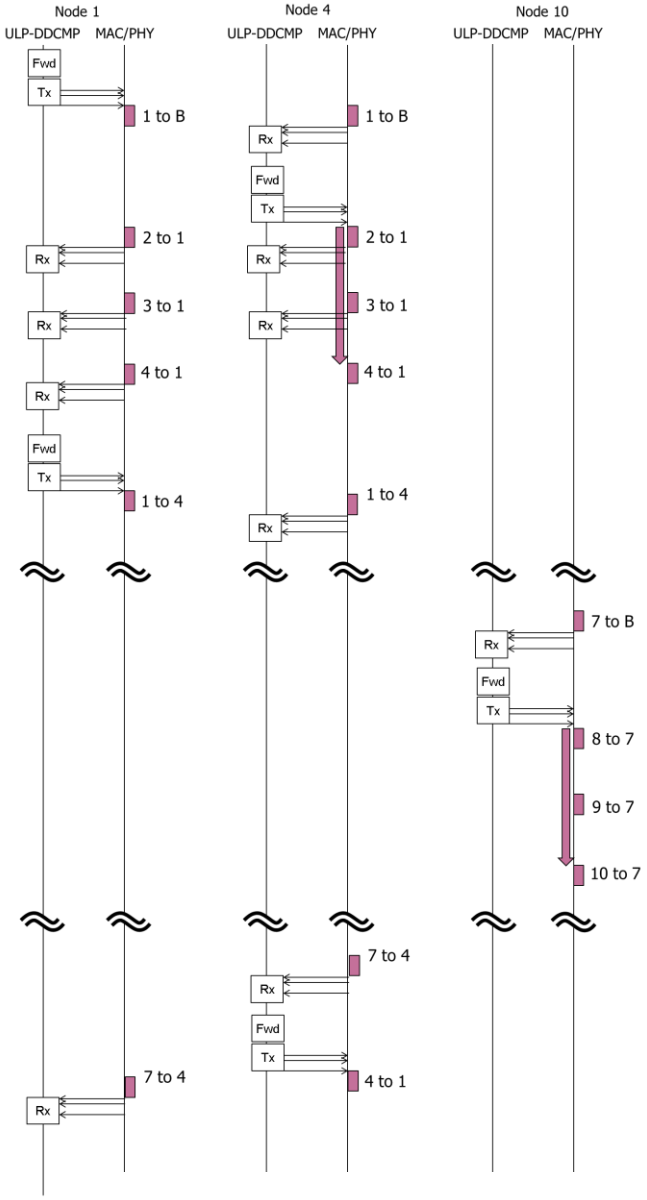


Fig. 3. Discovery phase

For these reasons, when a period of packet receiving process is shorter than a packet receiving interval, ULP-DDCMP does not enter a state of an overload.

4.3 Authentication phase

To consider packet flows, we assumed that node 1 obtains a certificate of node 5 (refer Fig. 4). And, processing flows on ULP-DDNS and sending and receiving packet at node 1, node 5, and node 2 which are relay nodes. Since node 1 and node 5 does not exist in each range of wireless access. The packets are transmitted by using LDCF. Colored rectangles represent periods of transmitted and received packets so as described above, and Rx, Fwd, Tx, and Auth represent periods of a the received packet process, the forwarding packet process, the transmitted packet process, and Authenticated process, respectively.

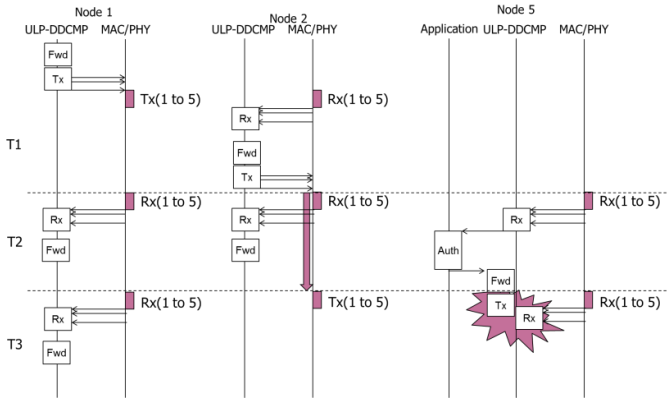


Fig. 4. Authentication phase

In T1, node1 judges that the acquisition of the certificate of node 5 is necessary from a trust relationship list, and node 1 send a packet to node 5. The packet is sent by using LDCF that is forwarding of flooding base because Node 5 is out of range of wireless area. The packet sent from node 1 is received at node2, node 3, and node 4. Node 2, node 3, and node4 judge that destination of the received packet is not local node. Each node starts a forwarding process and a transmission process to forward the packet.

In T2, node 3 that previously acquired the transmission right by CSMA/CA forwards the packet for the node 5 by using LDCF. This packet is also received at node 1, 2, 4, 5, 6. Node 2 and node 4 try to send the packet received in T1, because transmission timing was taken by node 3, the packet transmissions of node2 and node 4 are waited until next transmission timing in a MAC/PHY

Node2 that obtained transmission timing sends the packet of the node 5 addressing in T3, and the packet is received by node 1, 3, 4, and 5. In T2, node 5 which received the packet to local node has performed a process issuing the certificate. However, it is difficult to expect a processing completion time if this process is not performed with a data-driven processor. Thus, it cannot be guaranteed that a certificate issue processing, forwarding, and a transmission processing are completed by next packet reception. In the worst case, a packet might be arrived while processing forwarding or transmission. There is a possibility those processing and a packet receiving processing are executed at the same time.

When these are taken into consideration, a processing of requirement most strict in authentication phase is a process in T1 and in node 2. Thus, when a period of packet receiving process is shorter than a packet receiving interval, ULP-DDCMP does not enter a state of an overload.

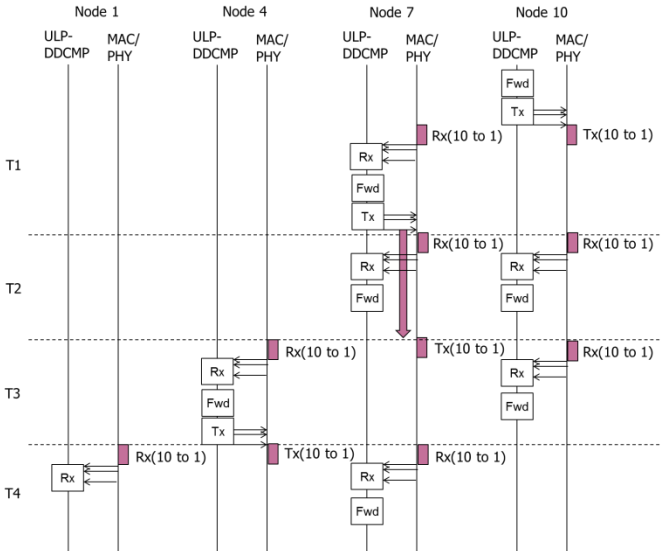


Fig. 5. Relay phase

4.4 Relay phase

A packet which is sent from node 10 is relayed in ad hoc network with LDCF and arrives in node 1. It is assumed that nodes relaying the packet are node 8, node 7, and node 4 until the packet arrives in node 1. Fig. 5 shows transmission and reception of packets and an internal processing of node 1, 4, 7, and 10. Node10 is a sender of information, node 1 is a receiver, and node 4 and node 7 are relay nodes as a representative.

At T1, a packet which will be sent from node 10 to the node 1 with flooding is received in node 7, and node 7 starts forwarding with LDCF.

At T2, node 8 sends a packet which has been received at T1 and is forwarded to node 1. Node 7 tries to send a packet, however, node 8 has sent the packet before, and the packet is stored in MAC/PHY until next transmission timing. Moreover, ULP-DDCMP has sent the packet addressed to node 1 to MAC/PHY at T1, as a packet that has already been processed. Only forwarding process is executed for the packet. A transmission process is not executed.

At T3, node 7 gets an opportunity to send the packet. Since the packet which has already been received, node 10 performs a forwarding process, however, node 10 does not perform a sending process. Node 4 starts a transmission process because node 4 received this packet for the first time. And, at T4, the node send a packet, node 1 receives the packet, and when a traffic of the packets addressed to node 1 which are flooded in ad hoc network disappears, a series of processing is completed.

Packet forwarding process according to reception of new packet similar to authentication phase is required most processing power in relay phase. Thus, when a total period of packet receiving process, forwarding process, and transmission process is shorter than a packet receiving interval, these processes does not overlap. By using ULP-DDNS, the period from start to completion of processing can be guaranteed, an overload can be avoided in advance.

5 Consideration

5.1 Overload avoidance scheme for sparse condition

Wireless communication is processed with one port generally. A sending packet and a receiving packet do not overlap unless a collision occurs. And, packet interval is also prepared. Therefore, shortest time until next packet is received can be predicted when the packet has been received. When each node works as a relay node, ULP-DDNS node executes a receiving process, a forwarding process, and a sending process and sends a packet according to following packet reception. It has been described that an overload is avoided when the receiving process, the forwarding process, and the sending process complete until next packet arrives. On the other hand, it was suggested that a receiving process and the other process may overlap in authentication phase. An overload avoidance scheme in case of a local node is source will be considered.

Basically, it cannot predict when a packet is received. Therefore, when local host sends a packet, forwarding process or transmission process and receiving process required according to packet reception might overlap. If a receiving process has started before a forwarding or transmission of which local host is source, the forwarding or transmission process can be delayed easily. In contrast, forwarding or transmission of which local host is source starts before a packet is received. Receiving process should be delayed, since refusing a receiving packet is the congestion itself. The overload avoidance scheme for an overlap of these processes is described in next sub section in detailed.

5.2 Overload avoidance scheme for dense condition

The case of one node in an ad hoc network acquiring information was examined. In actual case, since many nodes send and receive packets, a node might become in overload condition. Therefore, overload avoidance scheme in this situation was examined. If an internal process and a packet transmission process which are performed in local node delay, we can consider what is happen and can control the node. However, it is difficult to assume an influence when packet receiving and forwarding process delay. Therefore, to consider an overload avoidance scheme, the process is separated to a network process and an internal process. In this case, it is considered that the internal process can be delayed as described above, and a network processing is considered in this paper.

Table 1. Packet types

No.	Rx/Tx	Bcast/Ucast	From/To	denial	delay
1	Receive	Broad-cast	To local node	R	R
2		Broad-cast	To remote node	R	R
3		Uni-cast	To local node	R	R
4		Uni-cast	To remote node	R	R
5	Transmit	Broad-cast	From local node	R	A
6		Broad-cast	From remote node	A	R
7		Uni-cast	From local node	R	A
8		Uni-cast	From remote node	R	R

A: Allow
R: Refuse

There are two processes for network processing. One is a receiving process, and another is a transmitting process including a forwarding process. Moreover, there are a broadcast and a unicast for types of packet, and there are a local node and remote node for source and destination of a packet. These can be shown like Table. 1. A process that can be refused and a process that can be delayed are arranged for these network processing. And overload avoidance scheme is considered by refusing or delaying the process if ULP-DDNS node seems to be overload condition.

A transmitting and a receiving were examined at first in broad term. It is possible to delay or refuse the process so that a transmitting process is possible to confirm a content of a packet. However, it is impossible to delay or refuse a packet so that a receiving process cannot confirm a content of a packet until the process is completed

Next, decision from how each transmitting packet is used was tried. A packet of No. 5 is used as a source packet in a discovery phase or a source packet of LDCF forwarding. Since the packet is also used as reply for a request, it is possible to delay the packet processing. However, the processing is not permitted to be refused. A packet of No.6 is used at a relay node. Since the packet is sent to many nodes by flooding and LDCF has a mechanism to cancel a forwarding process depending on a processing load of a local node. The processing can be refused. Moreover, the delay is not permitted because there is a possibility that other nodes have already forwarded it and there is no meaning to delay processing. It is basically the same as No.5 though No.7 becomes the node is also source node in the unicast. No.8 is used for the unicast forwarding, and it is necessary to process in priority over other transmission processing because delaying this processing will greatly decrease the communication quality. Table 1 shows these summaries of the idea.

From these considerations, the transmission processing of No.6 is refused and the processing of No.5 and No.7 are delayed when the load of the node is high. In this case, because a unicast forwarding process is only performed as transmission process, an overload is not only avoided, but the increase of traffic is also avoided.

The delay and the refusal of the transmission process can be controlled by a local node. However, there is a situation that a receiving process must be delayed as described in the authentication phase. The local node cannot control a receiving packet, there is possibility that next packet cannot be processed by the delay of a receiving process. On the other hand, method controlling between nodes is proposed [6]. The method decreases a transmission rate of a remote node according to an allowable receiving rate noticed from peer node. The allowable receiving rate is obtained from an electrical current of ULP-DDCMP. When the receiving process delay occurs, an overload is avoided by a notice which lower value is less than a value obtained from current to reduce a transmission rate from a remote node temporary. The overload is avoided like this by the control of the transmission process delay in the local node and the control of the reception rate control between nodes.

6 Conclusion

scheme to avoid an overload of a network node required to achieve congestion-free ad hoc network communication was considered. Since a processing period of ULP-DDNS is constant even if multi-processing is required, it is possible to predict a period of processing. When ULP-DDNS is applied to urgent ad hoc network as an example, it was considered whether overload avoidance achieves by using the traffic pattern and packet types.

We have described an overload avoidance scheme as follows, (1) when the processing load becomes high, a transmission process is delayed before a receiving process, (2) Appropriate allowable throughput is notified to a remote node to lower the receiving packet rate when it is necessary to delay the receiving processing according to timing.

In future, we will evaluate the overload avoidance scheme and an appropriate scheme distributing tokens to each ULP-CUE will be examined.

Acknowledgement

Although it is impossible to give credit individually to all those who organized and supported the CUE project, the authors would like to express their sincere appreciation to all the colleagues in the project.

The CUE project is partially supported by Core Research for Evolutional Science and Technology (CREST), Japan Science and Technology Agency, SCOPE (Strategic Information and Communications R&D Promotion Programme), Ministry of Internal Affairs and Communications, Japan, the Grants-in-Aid for Scientific Research of Japan Society for the Promotion of Science and Semiconductor Technology Academic Research Center (STARC). And, this work is supported by VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc. and Cadence Design Systems, Inc.

Reference

[1] Keisuke Utsu, Naohide Fukushi and Hiroshi Ishii, "A Query-based Information Discovery method using Location Coordinates and its Contribution to Reducing Power Consumption in an Ad Hoc Network," Proc. of the 2010 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 610–615, July 2010.

[2] Keisuke Utsu, Hiroshi Sano, Turganzhan Kassymov, Hiroaki Nishikawa, and Hiroshi Ishii, "Proposal on Battery-aware Counterbased Flooding over Ad Hoc Networks," Proc. of the 2011 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP5141, July 2011.

[3] Hideaki Kawabata, Hiroshi Ishii, "Evaluation of Self-Organizing Key Management Framework Based on Trust Relationship Lists," Proc. Of the 2009 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 609–615, July 2009.

[4] Shuji Sannomiya, Ryotaro Kuroda, Kazuhiro Aoki, Kei Miyagi, Makoto Iwata, and Hiroaki Nishikawa, "Chip Multiprocessor Platform for Ultra-Low-Power Data-Driven Networking System: ULP-DDNS," Proc. of the 2011 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp.428–434, July 2011.

[5] Hiroaki Nishikawa, Kazuhiro Aoki, Hiroshi Ishii, and Makoto Iwata, "Intermediate Achievement of Ultra-Low-Power Data-Driven Networking System: ULP-DDNS," Proc. of the 2011 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp.421–427, July 2011.

[6] Hideki Yamauchi and Hiroaki Nishikawa, "Proposal of Applying ULP-DDNS to Congestion-Free Networking System," Proc. of the 2012 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP6031, July 2012.

Proposal of Applying ULP-DDNS to Congestion-Free Networking System

Hideki YAMAUCHI Hiroaki NISHIKAWA
Graduate School of Systems and Information Engineering, University of Tsukuba
Tsukuba Science City, Ibaraki, 305-8573 Japan

Abstract – *The importance of the ad hoc network is paid attention in order to secure communication under emergency or disaster. Traffic control in order to avoid the network congestion and to secure communication is also paid attention for same reason. However, traffic control method of packet communication on the ad hoc network has not been well established. Ultra Low Power Data Driven Networking System (ULP-DDNS) works based on the principle of data driven processing and achieves low power consumption networking system. The ad hoc network realized by the ULP-DDNS enables simple and effective node load measurement by power consumption and throughput linearity. Efficient traffic control method with the advantage to achieve congestion-free network is proposed.*

Keywords: Ad Hoc Network, Data-Driven, Congestion, Flow Control

1 Introduction

East Japan earthquake disrupted not only the public switched telephone network (PSTN), but also the Internet, and it caused congestion. However, because of the nature, while the throughput was very low, communication connection over the Internet was more secured and relatively better communication than PSTN. Moreover short message communication such as Twitter which can exchange information with less bandwidth was found applicable for minimum level of communication. For these reasons, IP communication was considered as an important means of securing emergency communication so that IP traffic control becomes more and more important.

In the East Japan earthquake, existing network infrastructure was swept by Tsunami and lost. When the infrastructure has been lost by such disaster, IP communication over ad hoc network has been attracting attention as a means of communication in order to secure emergency communications The communication in such situation, reachability to communication party is important, and to achieve effective communication, a certain level of other quality such as throughput and packet delivery ratio must be guaranteed as well. However, in the ad hoc network employed in an event of disaster is unable to have enough network capacity because of multi-hop communication and vulnerability caused by unstable power supply and node mobility or insufficient bandwidth. On the other hand, bandwidth demand such as large size video data transmission or data access concentration to specific information on servers is increasing even under disaster situation. For example, sharing a certain remote location situation with video and sharing necessary information on a server is popularly used. Therefore, appropriate traffic control is more and more important to avoid congestion caused by traffic concentration.

Traditional telephone network has traffic control system to avoid network congestion caused by traffic concentration such as ticket reservation, inquiries on the radio and TV or telephone of safety confirmation in the event of a large scale disaster such as catastrophic disasters of large earthquake and Tsunami in east Japan. The traffic control system detects the traffic concentration to a specific number or areas, or high traffic exceeding node capacity or trunk circuit capacity. Then, it limits or stops connecting the call to prevent network congestion and affection to not-related calls. The system is independent from network and centralized control is employed. It widely monitors network resource usage and loss of calls to detect congestion and control calls. Once congestion is detected, it controls switches preventing connection of calls to a congested number or area. However, such independent control system is not only increase cost and power consumption, but also it is not suitable for the ad hoc network whose nodes have more autonomy and distributed functionality. So, network system with autonomous and distributed traffic control capability which is applicable for ad hoc network has been desired.

Authors have been studying data-driven networking system, ULP-DDNS. Applying ULP-DDNS to the emergency ad hoc network under an emergency situation like loss of infrastructure by earthquake has also been studied to utilize its advantage of low power consumption.

This paper proposes a method of applying ULP-DDNS as a node to achieve congestion free emergency ad hoc network. It is distributed flow control of ad hoc network node to avoid congestion with maximizing throughput and network resource utilization without losing ULP-DDNS advantages.

2 Congestion and flow control

In this section, various congestion controls are described and their characteristics are discussed whether they are suitable to ULP-DDNS ad hoc network.

2.1 Existing congestion control mechanism

Congestion occurs when a network or a part of network have more traffic than its performance. Existing network congestion control system is to maintain traffic amount within network performance. From a network system point of view, there are three important parameters which represents network performance.

- Trunk bandwidth
- Node throughput(bandwidth)
- Switching rate

In the PSTN, traffic control system measures these performance indexes and related resource usage. When the system detects congestion occurrence, it commands originating switches to limit number of calls to the congested numbers, switches or areas. In the ATM network, traffic flow is managed by virtual circuit or virtual path bases. A traffic flow reserves network resources based on the requested bandwidth and the traffic is shaped and polished to be within the reserved bandwidth. Different from ATM and PSTN, IP packet network like the Internet does not carry out aggressive traffic control, but it simply discards overflowed packets. Traffic control is dependent on congestion avoidance scheme of TCP/IP which is widely deployed [1] [2] [3]. Also, if a new communication flow is added to network, it is difficult to avoid congestion reoccurrence.

Congestion avoidance of emergency ad hoc network is important to reduce unnecessary traffic and help reserving battery capacity when utility power is lost. Moreover, real time communication such as voice or video over IP network uses UDP which has no flow control mechanism. So that congestion avoidance dependent on TCP is insufficient for traffic control and is unable to avoid congestion completely.

2.2 Flow control for congestion avoidance

There are two approaches of flow control for congestion avoidance. They are open loop method and closed loop method. By the open loop method, a traffic source requests necessary bandwidth and traffic characteristics at beginning of transfer and network reserves necessary network resource and if the network resource is not enough, the request is rejected by the network. The source controls its traffic throughput within the bandwidth and characteristic. If the source violates their limit, the exceeding traffic is discarded. SBR and VBR of ATM are the example of the method. With the closed loop method, network monitors resource usage and feedbacks to the traffic source or transit nodes if traffic exceeds threshold. ABR of ATM or congestion control of PSTN belongs to the method. If the network resource is known, open loop is easier to guarantee quality of service (QoS). However, in the ad hoc network, especially when it is deployed for emergency, it is hard to assume that static network configuration is known. In addition, the network configuration dynamically changes due to new participants or out such as battery exhaust. So that closed loop control is more suitable to emergency ad hoc network because it is able to reflect network condition in real time.

There is another classification of flow control approach. One is controlling traffic after congestion occurrence and the other is controlling traffic before congestion occurrence. TCP/IP approach is former one. It is assumed that packet drop is due to network congestion and the discarded packet is retransmitted with fallen down rate to resume non-congestion communication. Even the network resume normal state, the retransmission consumes network resources additionally. On the other hand, flow control has a risk of over control. The network performance can be lower than its maximum allowance when traffic is not properly limited. However, this paper focuses on the flow control because it can achieve perfect congestion free.

Traffic control within network is not sufficient to avoid congestion. A traffic source may be requested to reduce or stop traffic eventually. End terminal must be included within flow control mechanism. Accurate flow control needs enough network resource usage or traffic load information in timely manner, however measuring them also consumes a certain amount of network resource. It is necessary to consider measurement and accuracy balance for cost and energy efficient traffic control.

2.3 Flow control implementation

The ad hoc network is composed of nodes which are autonomous and distributed nodes. In case of emergency, it is hard to implement flow control system in parallel with ad hoc network. Therefore, it is requested for the each ad hoc network nodes equipped with the flow control function control as well as traffic switching and routing functions.

Controlling traffic flow is necessary to know network load. In the emergency ad hoc network, it is not suitable to know whole network load or load of entire network nodes. On the other hand, flow control without other node load information is unable to prevent the node from causing other node congestion. As a result, at minimum, adjacency node load information is necessary to prevent neighbor node congestion. Such load information must be exchanged over the ad hoc network, because of its distributed and loose coupled nature. But the information exchange increases network load. Efficient method of information exchange is necessary. To control other node traffic directly is also not suitable for ad hoc network congestion control. Such control request must be passed via network connection if necessary.

In the emergency ad hoc network, the ULP-DDNS nodes are connected via wireless network. There are two types of congestion in the case. One is congestion within ULP-DDNS node which is caused by overload of resources in the node. To control the congestion it is necessary to measure resource

usage and mechanism to reduce the load using the resource. The other is wireless network congestion. The air section for wireless network is shared with neighbor nodes so that only one node is able to send data and other nodes which have data to be sent need to wait while the air is occupied. There is a possibility of congestion if neighbors have transmission requests same time. In this case a node is able to monitor air section usage because all node are able to monitor wireless signal same time except nodes out of range

3 Congestion in ULP-DDNS node

Figure 1 describes the block diagram of ULP-DDNS [4] which is developed for evaluation purpose. It has enough functions as an ad hoc network node. Congestion within the ULP-DDNS is discussed in this section and ULP-DDCMP, Communication Device and Application Processor are point of congestion and need a control for congestion avoidance.

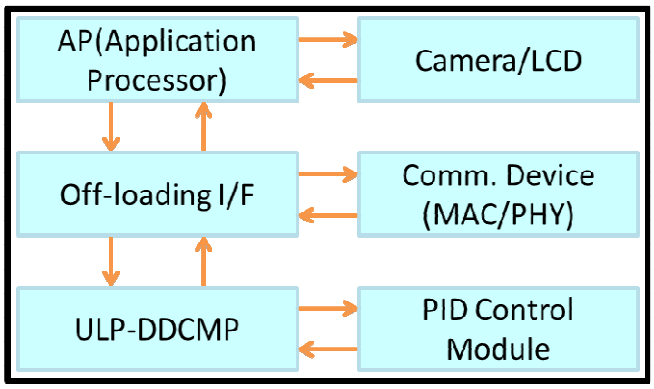


Figure 1 ULP-DDNS

3.1 Congestion in ULP-DDCMP

ULP-DDNS has an ultra-low-power data-driven chip-multi-processor: ULP-DDCMP whose block diagram is shown in Figure 2. The ULP-DDCMP functions as a network processor and an IEEE 802.11g wireless LAN interface is used as communication device. It also has an application processor, a camera and an LCD, they are usable for demonstration purpose of the ad hoc network application. The ULP-DDNS has enough functionality as an ad hoc network node.

Off-loading interface converts between tokens and data. Tokens are processed in ULP-DDCMP and data are processed in AP or transferred through communication device. With a data-driven principle, data arrives at ULP-DDCMP or in ULP-DDCMP begins to be processed when its data set is ready. After the data is processed, it leaves ULP-DDCMP and transferred to AP or communication device via off-loading interface.

ULP-DDCMP consists of 4 ULP-CUE data-driven processor cores. The ULP-CUE processes based on data-driven principle whose data and instruction sets flow in a packet format and it is called token in this paper. The tokens in the ULP-CUE are processed and sent to other ULP-CUE if necessary. Also, to communicate with outside device, Off-loading interface

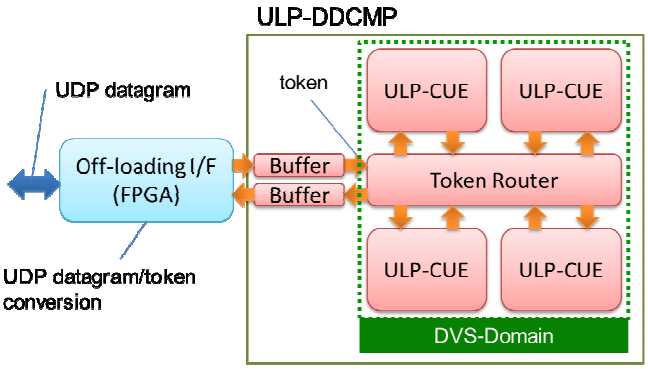


Figure 2 ULP-DDCMP

Figure 3 shows ULP-CUE block diagram which consists of pipeline of function stages. FC invokes a token which has a set of data and ready to be processed flows the circular pipeline. Instruction in a token is decoded at ID and executed at EX. The circular pipeline has a bypass circuit BB to MB for unary-operation which mating is unnecessary.

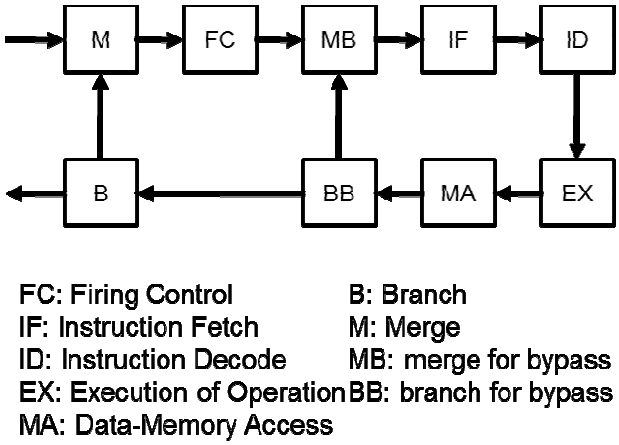


Figure 3 ULP-CUE Block Diagram

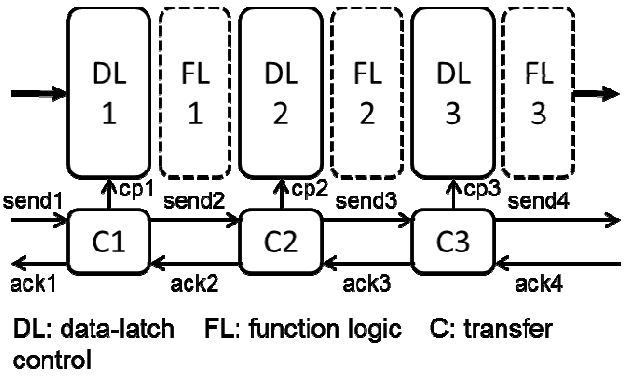


Figure 4 Self Timed Pipeline

The pipeline stages in the ULP-CUE are based on self-timed pipeline (STP) principle shown in Figure 4. A latched token is transferred to next data latch through function logic. Each latch is controlled by a transfer control which communicates with adjacency transfer control with handshake signal. Since the handshake is asynchronous and demand based, the STP realizes data driven principle. [5]

With above mentioned principle, ULP-CUE has almost flat turnaround time independent from number of threads as shown in Figure 5 and achieves scalability in throughput with number of processors. [6]

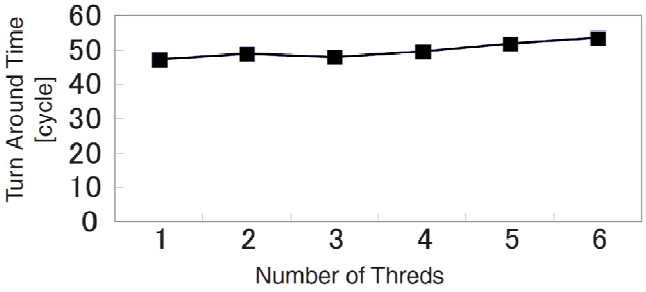


Figure 5 Number-of-Threads and TAT

With above mentioned principle, ULP-CUE has almost flat turnaround time independent from number of threads as shown in figure 5 and scalability in throughput with number of processors.

Figure 6 shows the relation between throughput and power consumption of ULP-STP. The ULP-STP has power gating (PG) function and the graph shows the relationship in three voltages. According to the result, power consumption and ULP-STP throughput relationship is almost in linear when voltage is same.

Processor load of ULP-DDCMP is measured by number of tokens processed and it is equivalent to packet throughput. Instead of counting the number of tokens in pipeline, it is able to measure processor load through power consumption with these characteristics of ULP-STP which compose ULP-DDCMP. Measuring power consumption is measuring electric current when voltage is same. Measuring current can be easily achieved without consume processing power of ULP-DDCMP. This enables easy load measurement without affecting processor performance. On the other hand, because TAT is almost flat, it is hard to assume load through TAT which is often used to measure processor load from outside.

From congestion control point of view, knowing other nodes load is necessary to control traffic. However, in the emergency ad hoc network, it is difficult to implement separate congestion control system so that nodes are requested to implement autonomous traffic control mechanism

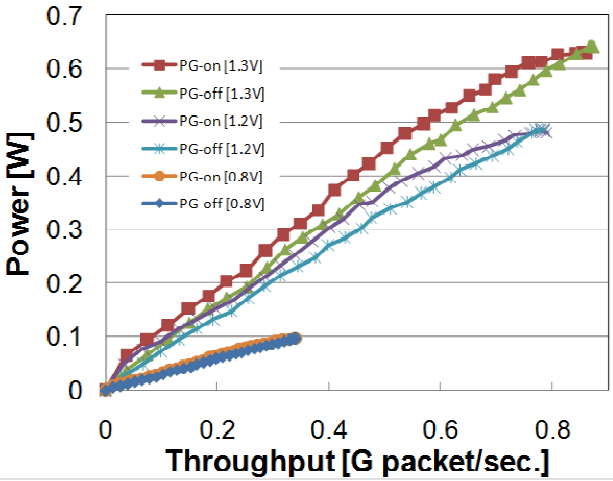


Figure 6 Throughput to Power Consumption

In this distributed traffic control implementation each node need to know load of other nodes, at least neighbor node load which the node communicates In order to know other node load, there are two strategies, measure node load from outside or announce self-measured load to other nodes. As mentioned above, TAT is useless but power consumption indicates load. However, it is difficult to measure power consumption from other nodes. Considering this, a ULP-DDNS is requested to announce its load to other nodes which is measured by its own power consumption.

3.2 Congestion of other blocks

There are other congestion of ULP-DDNS, high load of AP and communication device. AP congestion or high load and its load control are not discussed in this paper because it is an application level control and depends on the application in the AP. Although it is assumed that AP has However, the AP has to be informed available throughput of the network in order to avoid unnecessary traffic generation which can be waited or reduced resource allocation.

Another type of congestion exists in wireless network portion. The air section is shared with neighbor nodes and can be congested even the node load is low enough. If the air is busy because of the high usage, packet is queued and waiting until it is ready to be sent. If the waiting packet queue is already full, the queue will be dropped and discarded so that it is necessary to stop the packet queuing before the queue become full. The resource usage is not only immeasurable by ULP-DDCMP power consumption but also the power consumption decreases whichever the queue is implemented outside of ULP-DDCMP or within. If the queue is outside of ULP-DDCMP, its power consumption decreases when a packet leaves ULP-DDCMP. If the queue is implemented in ULP-DDCMP it means the packet related token(s) is stopped in the pipeline including FC so that power consumption of logic switching decreases. Therefore, the ULP-DDNS node

looks operating under low load when only power consumption is measured. It is necessary to implement the queue length counter for wireless network load measurement purpose.

4 Proposed flow control

4.1 Throughput margin announcement

As it is mentioned previously, ULP-DDNS node load is represented by power consumption of DDCMP and a queue length of the communication device. However, from other node point of view, the load information are useless because they do not explicitly tell the amount of data or throughput which they can transfer to the node. Although a ULP-DDCMP load may be useful for the ULP-DDNS node, the load is necessary to be converted into an index which represents available resource for other nodes. The queue length of communication device is also same. The load and queue length are different dimension but they are not completely independent. It is convenient for other nodes when these two indexes are merged into an index.

Following is the representative indexes popularly used for network system and useful for other nodes

- Available bandwidth (throughput)
- Available packet rate
- Available data size
- Available queue length

Bandwidth and packet rate are convertible when average packet size is fixed. Data size and queue length are also convertible because of same reason. Data size and bandwidth can be convertible when maximum burst rate and duration are defined as traffic characteristics. In this proposal, available bandwidth is selected as node load index because of following reason.

1. Bandwidth is physically defined and solid.
2. Packet length and necessary bandwidth depend on the protocol even if pay load data size is the same.
3. Bandwidth is popular and often used for performance index

Available network throughput $th_{avail\_ulp-ddcmp}(V,I)$ is calculated from available ULP-DDCMP throughput measured by volts and current. A conversion ratio of average UDP/IP packet size and average number of tokens is necessary. This depends on program implementation on ULP-DDCMP, its parallelism and stages.

There is another throughput margin of queue of communication device. When ULP-DDNS communicates with other node, it is via communication device and the throughput consists of the communication device throughput and air section throughput. To use the communication device, there is a queuing buffer to absorb the throughput difference. The throughput of the queue $Th_{avail\_comm.dev}(Q)$ is described as following

1. No queue: the throughput is equal to or less than network throughput and there is throughput margin to be increased.
2. The queue length is constant: the throughput is equal to network throughput and there is no throughput margin.
3. Increasing queue length: the throughput is higher than network throughput and throughput must be decreased until the queue length decreasing. Throughput margin is negative.
4. Decreasing queue length: the throughput is lower than network throughput and keep the throughput until the queue disappear. The throughput margin is 0.

In the ULP-DDNS, ULP-DDCMP and communication device are series connected as it is shown Figure 7. In the model, throughput is limited by the slowest one so that announced is throughput $th_{available}$ is

$$th_{available} = \min(th_{avail\_ulp-ddcmp}(V,I), th_{avail\_comm.dev}(Q)) \quad (1)$$

Figure 7

The ULP-DDNS ad hoc network employs wireless communication so that the throughput margin can be broadcasted to neighbor nodes within transmission range. The broadcasted announcement helps reduce time and power of node. On the other hand, nodes which can receive the announcement are requested to monitor and check it for next transfer. Otherwise, it has to wait until next announcement.

4.2 Simultaneous transmission and throughput allocation

Simple data transfer control mentioned above has a simultaneous transmission problem shown in Figure 8. If a node which receives an announcement can transfer within the announced throughput, total throughput may exceed the announced throughput. To avoid the problem, throughput allocation protocol is necessary.

For example, when Node 1 announces available throughput 1Mb/s and Node 2, Node 3 and Node 4 has 500kb/s throughput request for each. The simultaneous transfer cause overload of Node 1 because total throughput is 1.5Mb/s. To avoid the congestion, Node 1 have to permit or deny transfer request each by each or reallocate lower throughput to keep total throughput being available throughput. However, generally speaking Node 1 does not know characteristic of data transfer except it is clearly mentioned so that whether other nodes can accept lower throughput or not is not guaranteed. Considering this, ULP-DDNS node also need throughput allocation mechanism with permit or deny basis.

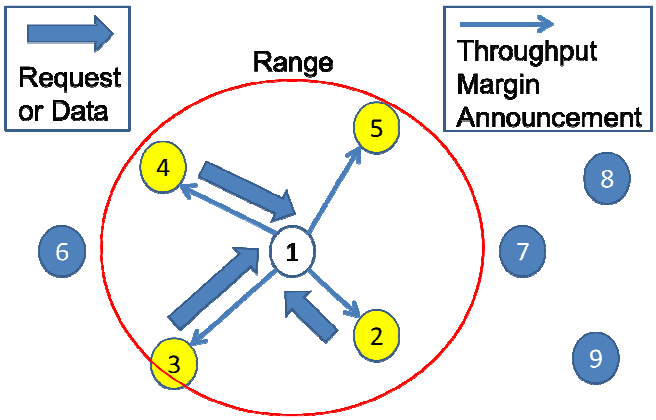


Figure 8 Simultaneous transmission

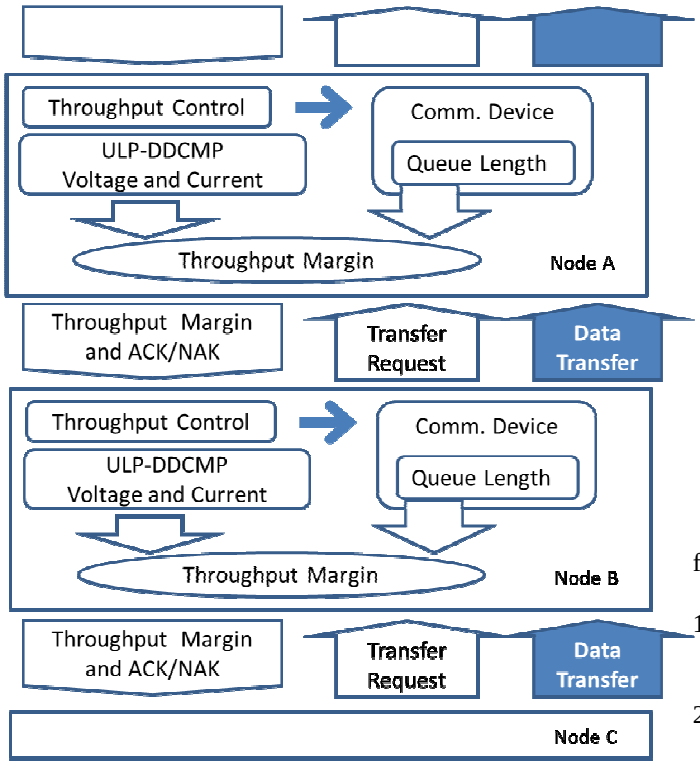


Figure 9 Proposed flow control mechanism

4.3 Flow control mechanism and protocol

In the communication, there are two type of traffic. One is not always expected to be reached. Like a broadcast, the traffic may be discarded when receiver is busy or off. The other is communication which needs a certain level of reachability and may need to know who the receiver is the later type of communication is the first priority for congestion free network. Although the broadcast type of traffic is considered as "load", flooding traffic is not described in this protocol. It is assumed that flooding traffic may not be received or discarded by node whenever a node is in high load or occupied.

Flow control functional model is shown in Figure 9 and Figure 10 describes the proposed flow control protocol sequence. Figure 9 describes relationship of nodes in series. A node may have other relationship such as a hub-spoke model

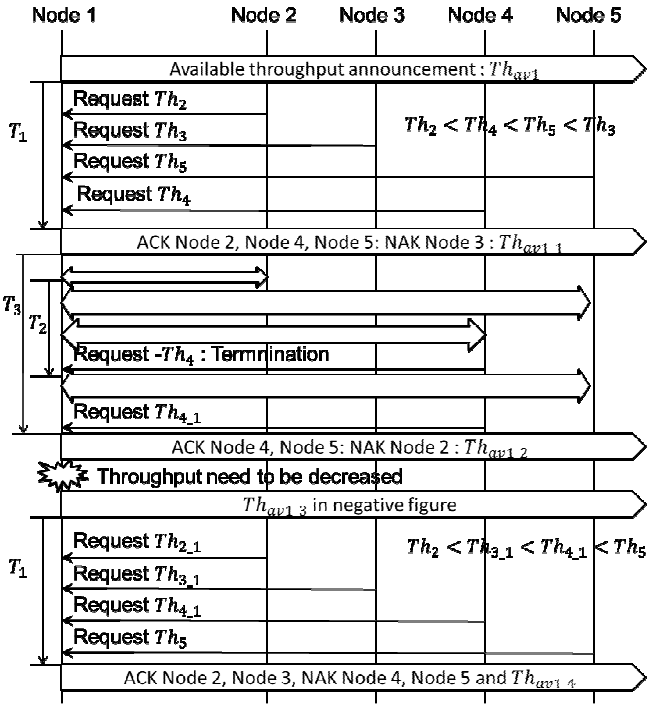


Figure 10 Proposed flow control protocol

The flow control protocol described in Figure 10 is following. Example in Figure 10 is written in brackets.

1. An ad hoc network node (Node 1) has a function to measure its throughput margin based on voltage and current of ULP-DDCMP and queue length of communication device.
2. Throughput margin is calculated from them. It is announced in broadcast (Th_{av1})
3. Among the neighbor nodes (Node 2, Node 3, Node 4 and Node 5) which have transmission request to Node 1 send transmission request with Th_n (Node 2, Node 3 and Node 4 ; $Th_2 < Th_4 < Th_5 < Th_3$) within the timer T_1.
4. If there is no request, the node (Node 1) announce again after T_3
5. The node (Node 1) accepts traffics in the order of lower to higher while $\sum Th_n$ is less than Th_{av1} and if it exceeds Th_{av1} the request is denied.(Node 2 and Node 4 are accepted and Node 5 and Node 3 are denied)
6. The node (Node 1) announces new available throughput (Th_{av_1}). The announcement is with ACK list of accepted request and NAK list of denied request (ACK Node 2, Node 4and Node 5: NAK Node 3)
7. Each node begins data transfer within the requested throughput.

8. When data transfer finishes, a node notifies termination with transfer request termination with the node throughput in negative figure (Node 4)
9. While a node continuously transfers data within T_2 it is regarded as sending request 0 (Node 5)
10. When there is no transfer within T_2 and no transfer until announcement, it is regarded as termination of transfer (Node 2).
11. A node is able to make new request even just after termination. (Node 4 with Th_{4_1}) It is also able to request to decrease throughput with negative figure.
12. After T_3 the node announces new available throughput (Th_{av1_2}) with all ACK list of accepted requests and NAK list of denied requests. (ACK Node 4 and Node 5: NAK Node 2)
12. T_2 time out flow is explicitly listed in NAK (Node 2) in order to inform termination.
13. When the node is requested to decreased due to the node internal reason, it announces available throughput in negative figure in order to request nodes to reduce
14. All nodes (Node 4 and Node 5) suspend their transfer upon negative available throughput announcement.
15. Each node decide new request referring requested throughput decrease. The new requests are expected to be decreased. The same throughput request is also acceptable although the request has higher risk of being denied.
15. "Same procedure in 3 and 4." Each nodes send transmission request with new throughput (Th_{2_1}, Th_{3_1}, Th_{4_1} *and* Th_5) within the timer T_1. If there is no request, the node (Node 1) announce again after T_3
16. "Same as procedure in 5." The node (Node 1) chooses request. (Node 2 and Node 3 are accepted, and Node 4 and Node are denied)
16. "Same procedure in 6." The node (Node 1) announces new available throughput (Th_{av1_4}). The announcement is with ACK list of accepted requests and NAK list of denied requests (ACK Node 2 and, Node 3: NAK Node 4 and Node 5)

5 Conclusion

This paper described congestion in the emergency ULP-DDNS ad hoc network and proposes a method of congestion avoidance to achieve congestion free network. The proposed flow control is based on distributed control therefore independent flow control system is unnecessary. It is proposed simple and non-load measurement with utilizing the ULP-DDCMP advantage whose load is measurable with its power consumption. It also takes in account of sending queue. The flow control protocol considering distributed implementation is also proposed.

The proposed measurement has assumption that the number of tokens in the ULP-DDCMP and UDP/IP packet traffic is in proportion. It is also assumed that communication device queue and throughput is convertible in certain ratio. Further study and experiment is needed in this area.

Throughput control described in the paper is not yet implemented. Although the traffic shaping mechanism is widely implemented in many network components, a study may be requested for data-driven shaper implementation.

6 Acknowledgements

Although it is impossible to give credit individually to all those who organized and supported the CUE project and the ULP-DDNS project, the authors would like to express sincere appreciation to all members in the project.

7 References

[1] S. Floyd and K. Fall., "Promoting the Use of End-to-End Congestion Control in the Internet", IEEE/ACM Transactions on Networking, August 1999

[2] M. Allman, V. Paxson, W. Stevens, "RFC2581 - TCP Congestion Control " , IETF, April 1999

[3] J. Postel, "RFC793 – Transmission Control Protocol", IETF, Sep 1981

[4] H. Nishikawa, K. Aoki, H. Ishii and M. Iwata, "Intermediate Achievement of Ultra-Low-Power Data-Driven Networking System: ULP-DDNS", Proc. of the 2011 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 421-427, July 2011.

[5] S. Sannomiya, R. Kuroda, K. Aoki, K. Miyagi, M. Iwata and H. Nishikawa, "Chip Multiprocessor Platform for Ultra-Low-Power Data-Driven Networking System: ULP-DDNS", Proc. of the 2011 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 428-434, July 2011.

[6] H. Nishikawa, H. Tomiyasu, M. Okamoto, M. Sugiyama, H. Uchida, O. Mizuno, H. Ishii and M. Iwata, "CUE-v3: Data-Driven Chip Multi-Processor for Ad hoc and Ubiquitous Networking Environment", Proc. of the 2007 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp.623-629, July 2007.

[7] Y. Nishida and H. Nishikawa, "A Study on Overload-Avoidance Scheme of ULP-DDNS for Congestion-Free Networking System", Proc. of the 2012 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP6030, July 2012

SESSION

PARALLEL AND DISTRIBUTED ALGORITHMS AND APPLICATIONS

Chair(s)

TBA

Study on Parallel SVM Based on MapReduce

Zhanquan Sun[1], **Geoffrey Fox**[2]
[1] Key Laboratory for Computer Network of Shandong Province, Shandong Computer Science Center, Jinan, Shandong, 250014, China
[2]School of Informatics and Computing, Pervasive Technology Institute, Indiana University Bloomington, Bloomington, Indiana, 47408, USA

Abstract - *Support Vector Machines (SVM) are powerful classification and regression tools. They have been widely studied by many scholars and applied in many kinds of practical fields. But their compute and storage requirements increase rapidly with the number of training vectors, putting many problems of practical interest out of their reach. For applying SVM to large scale data mining, parallel SVM are studied and some parallel SVM methods are proposed. Most currently parallel SVM methods are based on classical MPI model. It is not easy to be used in practical, especial to large scale data-intensive data mining problems. MapReduce is an efficient distribution computing model to process large scale data mining problems. Some MapReduce software were developed, such as Hadoop, Twister and so on. In this paper, parallel SVM based on iterative MapReduce model Twister is studied. The program flow is developed. The efficiency of the method is illustrated through analyzing practical problems.*

Keywords: Parallel SVM, Large scale data, MapReduce, Twister

1 Introduction

With the development of electronic and computer technology, the quantity of electronic data is in exponential growth [1]. Data deluge has become a salient problem to be solved. Scientists are overwhelmed with the increasing amount of data processing needs arising from the storm of data that is flowing through virtually every science field, such as bioinformatics [2-3], biomedical [4-5], Cheminformatics [6], web [7] and so on. Then how to take full use of these large scale data to support decision is a big problem encountered by scientists. Data mining is the process of discovering new patterns from large data sets involving methods at the intersection of artificial intelligence, machine learning, statistics and database systems. It has been studied by many scholars in all kinds of application area for many years and many data mining methods have been developed and applied to practice. But most classical data mining methods out of reach in practice in face of big data. Computation and data intensive scientific data analyses are increasingly prevalent in recent years. Efficient parallel/concurrent algorithms and implementation techniques are the key to meeting the scalability and performance requirements entailed in such large scale data mining analyses. Many parallel algorithms are implemented using different parallelization techniques such as threads, MPI, MapReduce, and mash-up or workflow technologies yielding different performance and usability characteristics [8]. MPI model is efficient in computation intensive problems, especially in simulation. But it is not easy to be used in practical. MapReduce is a cloud technology developed from the data analysis model of the information retrieval field. Several MapReduce architectures are developed now. The most famous is the Google, but the source code is not open. Hadoop is the most popular open source MapReduce software. It has been adopted by many huge IT companies, such as Yahoo, Facebook, eBay and so on. The MapReduce architecture in Hadoop doesn't support iterative Map and Reduce tasks, which is required in many data mining algorithms. Professor Fox developed an iterative MapReduce architecture software Twister. It supports not only non-iterative MapReduce applications but also an iterative MapReduce programming model. The manner of Twister MapReduce is "configure once, and run many time" [9-10]. It can be applied on cloud platform. It will be the popular MapReduce architecture in cloud computing and can be used in data intensive data mining problems.

Support Vector Machines are powerful classification and regression tools [11]. Many SVM software models have been developed, such as libSVM, lightSVM, ls-SVM and so on. LibSVM is taken as the most efficient SVM model and widely applied in practice because of its excellent property [12]. But SVM's compute and storage requirements increase rapidly with the number of training vectors, putting many problems of practical interest out of their reach. The core of an SVM is a quadratic programming problem (QP), separating support vectors from the rest of the training data. For improving the training speed of SVM, many efforts have been done. Reference [13] accelerates the QP with 'chunking', where subsets of the training data are optimized iteratively, until the global optimum is reached. Sequential Minimal Optimization (SMO) [14], which reduces the chunk size to 2 vectors, is the most popular of these algorithms. Eliminating non-support vectors early during the optimization process is another strategy that provides substantial savings in computation. Parallelization has been proposed by splitting the problem into smaller subsets and training a network to assign samples to different subsets [15]. Variations of the standard SVM algorithm, such as the Proximal SVM have been developed that are better suited for parallelization [16], but how widely they are applicable, in particular to high-

dimensional problems, remains to be seen. A parallelization scheme was proposed where the kernel matrix is approximated by a block-diagonal [17]. Most of parallel SVM are based on MPI programming model. Little research work has been done with MapReduce work.

Based on current research work of SVM and Twister MapReduce framework, the paper develops a parallel SVM model based on MapReduce. In this model, training samples are divided into subsections. Each subsection is trained with a SVM model. In this paper, libSVM is used to train each subSVM. The non-support vectors are filtered with subSVMs. The support vectors of each subSVM are taken as the input of next layer subSVM. The global SVM model will be obtained through iteration. The MapReduce based SVM model is encoded with Java language.

The following of the paper is organized as follows. LibSVM method is introduced briefly in part 2. The Twister model is introduced in part 3. MapReduce based parallel SVM model and its program flow is introduced in part 4. Two practical examples are analyzed with the proposed model in part 5. At last some conclusions are summarized.

2 LibSVM

2.1 Support Vector Machines

SVM first maps the input points into a high-dimensional feature space with a nonlinear mapping function Φ and then carries through linear classification or regression in the high-dimensional feature space. The linear regression in high-dimension feature space corresponds to the nonlinear classification or regression in low-dimensional input space. The general SVM can be described as follows.

Let l training samples be $T = \{(x_1, y_1), \cdots, (x_l, y_l)\}$, where $x_i \in R^n$, $y_i \in \{1, -1\}$ (classification) or $y_i \in R$ (regression), $i = 1, \cdots, l$. Nonlinear mapping function is $k(x_i, x_j) = \emptyset(x_i)\emptyset(x_j)$. Classification SVM can be implemented through solving the following equations.

$$\min_{w,\xi_i,b}\left\{\frac{1}{2}\|w\|^2 + C\sum_i \xi_i\right\}$$
$$s.t.\, y^i(\Phi^T(X_i)w + b) \geq 1 - \xi_i \quad \forall i = 1, \cdots, n \qquad (1)$$
$$\xi_i \geq 0 \quad \forall i = 1, \cdots, n$$

By introducing Lagrangian multipliers, the optimization problem can be transformed into its dual problem.

$$\min_{\alpha} \sum_{i,j} \alpha_i \alpha_j y_i y_j k(x_i, x_j) - \sum_{i=1}^{l} \alpha_i$$
$$s.t.\ y^T \boldsymbol{\alpha} = 0 \qquad (2)$$
$$0 \leq \alpha_i < C, i = 1, \cdots, l$$

After obtaining optimum solution a^*, b^*, the following decision function is used to determine which class the sample belongs to.

$$f(x) = sgn\left(\sum_{i=1}^{l} y_i \alpha_i^* K(x_i, x) + b^*\right) \qquad (3)$$

The classification precision of the SVM model can be calculated as

$$\text{Accuracy} = \frac{\#\text{correctly predicted data}}{\#\text{total testing data}} \times 100\%$$

2.2 libSVM

LibSVM is taken as the most efficient SVM software. It is an integrated software for support vector classification, regression, and distribution estimation. Most efficient analysis models are included. For example, C-SVC and nu-SVC classification models, epsilon-SVR and nu-SVR regression models, and one-class SVM distribution estimation. For improving the classification correct rate, cross validation is adopted. For processing unbalancing classification problem, weighted and probability models are adopted. The detail of libSVM can be found in [12]. In this paper, C-SVC libSVM model is selected to analyze the classification problems.

3 Architecture of Twister

There are many parallel algorithms with simple iterative structures. Most of them can be found in the domains such as data clustering, dimension reduction, link analysis, machine learning, and computer vision. These algorithms can be implemented with iterative MapReduce computation. Professor Fox developed the first iterative MapReduce computation model Twister. It has several components, i.e. MapReduce main job, Map job, Reduce job, and combine job. Twister's programming model can be described as in figure 1.

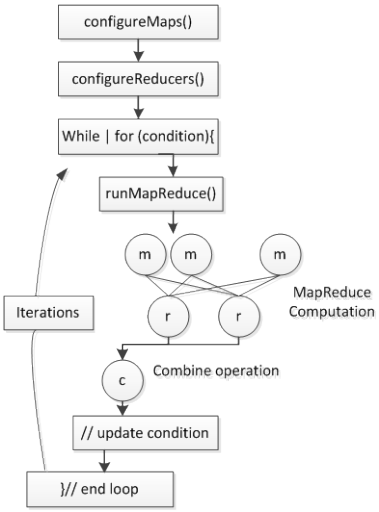


Fig. 1 Program model of Twister

MapReduce jobs are controlled by the client node through a multi-step process. During configuration, the client assigns MapReduce methods to the job, prepares KeyValue pairs and prepares static data for MapReduce tasks through the partition file if required. Between iterations, the client receives results collected by the Combine method, and, when the job is done, exits gracefully. The message communicate between job is realized with message brokers, i.e. NaradaBrokering or ActiveMQ.

Map daemons operate on computation nodes, loading the Map classes and starting them as Map workers. During initialization, Map workers load static data from the local disk according to records in the partition file and cache the data into memory. Most computation tasks defined by the users are executed in the Map workers. Twister uses static

scheduling for workers in order to take advantage of the local data cache. In this hybrid computing model, daemons communicate with the client through messages.

Reduce daemons operate on computation nodes. The number of reducers is prescribed in client configuration step. The reduce jobs depend on the computation results of Map jobs. The communication between daemons is through messages.

Combine job is to collect MapReduce results. It operates on client node. Twister uses scripts to operate on static input data and some output data on local disks in order to simulate some characteristics of distributed file systems. In these scripts, Twister parallel distributes static data to compute nodes and create partition file by invoking Java classes. For data which are output to the local disks, Twister uses scripts to gather data from all compute nodes on a single node specified by the user.

4 Parallel SVM based on Twister

4.1 Architecture of Parallel SVM

The parallel SVM is based on the cascade SVM model. The SVM training is realized through partial SVMs. Each subSVM is used as filter. This makes it straightforward to drive partial solutions towards the global optimum, while alternative techniques may optimize criteria that are not directly relevant for finding the global solution. Through the parallel SVM model, large scale data optimization problems can be divided into independent, smaller optimizations. The support vectors of the former subSVM are used as the input of later subSVMs. The subSVM can be combined into one final SVM in hierarchical fashion. The parallel SVM training process can be described as in figure 2.

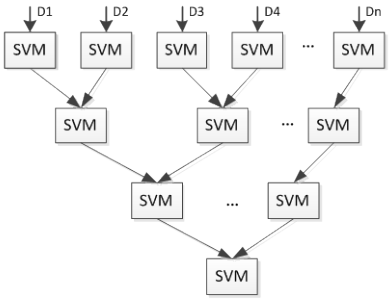


Fig. 2 training flow of parallel SVM

In the architecture, the sets of support vectors of two SVMs are combined into one set and to be input a new SVM. The process continues until only one set of vectors is left. In this architecture a single SVM never has to deal with the whole training set. If the filters in the first few layers are efficient in extracting the support vectors then the largest optimization, the one of the last layer, has to handle only a few more vectors than the number of actual support vectors. Therefore, the training sets of each sub-problems are much smaller than that of the whole problem when the support vectors are a small subset of the training vectors. In this paper, libSVM is adopted to train each subSVM.

4.2 Program flow

From the parallel SVM architecture, the pseudo program code based on Twister is as follows.

```
Preparation
    Computation environment configuration
    Data partition and distribution to the computation nodes
    Create partition file
Main class
    JobConf; //configure the MapReduce parameters and classnames
    TwisterDriver; //to initiate the MapReduce tasks
    While(condition)  //not combined to one SVM
      JobConf;  //reconfigure the MapReduce parameters;
      TwisterDriver; // initiate new MapReduce tasks, Broadcast combined
    support vectors to each computation node;
      Get feedback results;
    If(condition) break; // if one SVM obtained, program finished
End mian class
Map class
    If(the first layer SVM)
      Load data from local file system;
    else
      Read data broadcasted by Main class
    End if
    Svm_train(); //the parameters of the SVM model are transformed through
    jobConf.
    Collector;  //sent the training result to Reduce job through message.
End Map class
Reduce class
    Read data transformed from Map job;
    Combine support vectors of each two subSVM into one sample set.
    Collect; //feedback all the trained support vectors
End Reduce class
```

Firstly, computation nodes should be available. The program can be described as follows. Original large scale data D should be partitioned into smaller data sections $\{D_1, \cdots, D_n\}$. These data sections are put to computation nodes. Then create partition file according to Twister command. The partition file will be used in Twister configuration.

Based on the available computation environment, jobConf is used to configure the computation parameters, such as Map, Reduce, and Combine class names, number of Map tasks and Reduce tasks, partition file and so on. TwisterDriver will initiate the MapReduce task. Dynamic parameters will be transformed to each computation node through API interface.

In each computation node, Map tasks are operated. In the first layer of figure 2, sample data are loaded from local file system according to partition file. In the following layers, the training samples are support vectors of former layer. LibSVM is used to train each subSVM. In the LibSVM, Sequential Minimal Optimization is used to select the workset in decomposition methods for training support vector machines [14]. C-SVC model is used to train classification SVM. Trained support vectors are sent to the Reduce jobs.

In the Reduce job, all support vectors of all Map jobs are collected together and feed back to client. Through iteration, the training process will stop when all subSVM are combined to one SVM.

4.3 Computation time analysis

The time cost of SVM can be divided into following sections. The computation time complexity of libSVM is $O(n^2)$. The transformation time of data between Map and Reduce nodes is depend on the bandwidth of the connection network. The transfer time can be described as t_{trans}. The combination time cost of two SVMs is O(n). When training data set is divided into m partitions, the computation cost is calculated as follows. The layers of cascade SVM is $N = \log_2 m$. Suppose that the ratio between the number of support vectors and that of whole training sample is α ($0 < \alpha < 1$) and the ratio between support vectors and that of training sample except the first layer is $\beta(1 < \beta < 2)$, i.e. the number of the last layer in Fig. 2 is $n_N = n\alpha\beta$ and the number of training sample of the first layer is almost $n_1 = n/m$. The number of training samples of the i layer is $n_i = n\alpha\beta * \left(\frac{\beta}{2}\right)^{N-i}$. So the computation time can be calculated as follows.

$$t = O\left(\left(\frac{n}{m}\right)^2\right) + \sum_{i=N}^{2} O\left(\left(n\alpha\beta * \left(\frac{\beta}{2}\right)^{N-i}\right)^2\right) + O\left(\sum_{i=N-1}^{2} n\alpha\beta * \left(\frac{\beta}{2}\right)^{N-i} * 2^{N-i}\right) + t_{trans} \quad (4)$$

Overhead of data transfer mainly includes three parts. The first part is data transfer from Maptask nodes to Reducetask nodes. The transferred data are the support vectors obtained by Maptask nodes. The second part is data transfer from Reducetask nodes to server node. The transferred data is the support vectors also. The third part is the data transfer from server nodes to Maptask node. The transferred data is the training samples combined by two subSVM's support vectors. The overhead of data transfer depend on the bandwidth of the MapReduce cluster.

From the architecture of parallel SVM, we can find that it is hierarchal structure. The low level SVM training has to be performed when all the upper level subSVM be trained. In the last level of the architecture, all the support vectors should be included in the training samples. The sample size must be bigger than the number of support vectors. When the ratio between support vector and training sample is bigger the speed up will be less. It is the shortcoming of the cascade SVM model.

5 Examples

All examples are analyzed in India cluster node of FutureGrid. Eucalyptus platform is adopted to configure the MapReduce computation environment. Twister0.9 software is deployed in each computation nodes. ActiveMQ is used as message broker. The configuration of each virtual machine is as follows. Each node is installed Ubuntu Linux OS. The processor is 3GHz Intel Xeon with 10GB RAM.

5.1 Adult data analysis

5.1.1 Data source

The source data are downloaded from NEC laboratory American Inc. website http://ml.nec-labs.com/download/data/milde/. In the adult database, 123 attributes are labeled 2 classes. Each attribute denoted by binary variable, i.e. 0 or 1. Labels are denoted by +1 or -1. It is a binary classification problem. The database includes two files. One is used for training and the other is used for testing. The training file includes 32562 samples. The testing file includes 16282 samples. In this example, 5 computational nodes are used. Training data are partitioned into n sections randomly. Each section has roughly equal number data.

5.1.2 Training process

The problem is taken as a binary classification problem. C-SVC model is adopted. The parameter of the SVM model is set as follows. Constant C is set 1, radial basis function is taken as kernel function, and gamma is set as 0.01. Firstly, the example is analyzed with only 1 computation node, i.e. classical SVM method is used to train the SVM model. The trained model is used to predict the testing samples. The training time and classification correct rate are listed in Table 1. Secondly, the example is analyzed with the parallel SVM based on map/reduce. For comparison, the sample is partitioned into 2, 3, 4, 5 sub-samples respectively. When the sample is partitioned into 2 sub-samples, 2 computing nodes are used. The training time and classification rate of each partition form are listed in table 1. And so forth to the other partitions.

Table 1 analysis result of SVM with different partition nodes

Number of nodes	Number of SVs	Training time(s)	Classification correct rate
1	11957	490.591	84.82
2	11933	281.152	84.98
4	11908	239.914	83.06
8	11887	237.441	82.74

5.1.3 Feature selection with correlation coefficient

In the example, there are 123 attribute variables. Most variables provide minor contribution in the classification. For improve the training speed and reduce the noise effect of attribute variables, correlation coefficient is used to measure the correlation between class variable Y and attribute variable X. The attribute variables are selected according to the correlation values. The correlation coefficient is calculated as the following equation.

$$\rho_{X,Y} = \frac{cov((X,Y)}{\sigma_X \sigma_Y} = \frac{E[(x-\mu_X)(y-\mu_Y)]}{\sigma_X \sigma_Y} \quad (5)$$

where $cov((X,Y)$ is the covariance of the two variables, σ_X, σ_Y are the standard deviations of X and Y. After calculating the correlation coefficient, the pruning value is set 0.1. At last, 34 attribute variables are selected. The training result is listed in table 2.

Table 2 analysis result of SVM based on feature selection

nodes Number	Number of SVs	Training time(s)	Classification correct rate
1	11702	154.098	84.10
2	11694	94.338	84.06
4	11710	86.142	83.95
8	11692	83.57	82.99

5.1.4 Results analysis

The analysis results are shown as in Fig.3 and Fig. 4. From Fig.3 we can find that the training time can be reduced

greatly when the sample is partitioned 2 parts. But with the increase of partition number, the training time reduction will become slow. From Eq. (8), the computation cost mostly concentrate on the training calculation of each subSVM. The example was analyzed in HPC cluster. The data transfer time cost is minor. In this example, the ratio $\alpha \approx 0.35$ and $\beta \approx 1.2$. The last layer will occupy the mainly part computation time and it will not decrease with the increase of partition number. With the decrease of α, the computation time can be reduced more. With the introduction of feature selection, the computation can be reduced greatly without decreasing the correct classification rate.

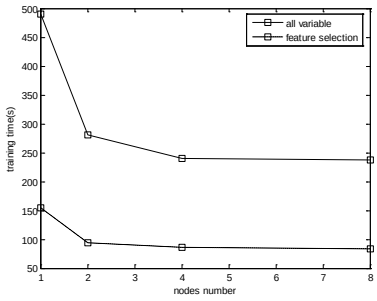


Fig. 3 Training time based on different partition nodes

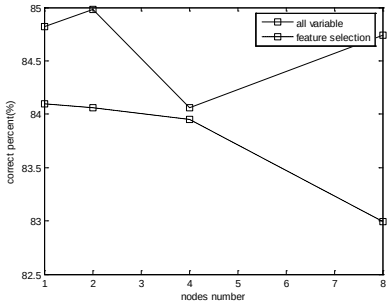


Fig. 4 Correct rate based on different partition

5.2 Forest Cover type Classification

5.2.1 Data source

The source data are downloaded from http://ftp.ics.uci.edu/pub/machine-learningdatabases/covtype/. The data is used to classify forest cover type. The original data are collected by Remote Sensing and GIS Program, Department of Forest Sciences, College of Natural Resources, Colorado State University. Natural resource managers responsible for developing ecosystem management strategies require basic descriptive information including inventory data for forested lands to support their decision-making processes. The purpose is to predict the forest cover type according to cartographic variables' values. The square of each observed section is 30 x 30 meter cell. There are 54 columns in each data item. They denote 12 variables, i.e. Elevation, Aspect, Slope,Horizontal_distance_to_hydrology,Vertical_Distance_To_Hydrology,Horizontal_Distance_To_Roadways,Hillshade_9am,Hillshade_Noon,Hillshade_3pm,Horizontal_Distance_To_Fire_Points, Wilderness_Area, and Soil_Type, where Wilderness_Area is denoted by 4 binary columns and Soil_Type is denoted by 40 binary columns. They are labeled as 7 cover types, i.e. Spruce/Fir, Lodgepole Pine, Ponderosa Pine, Cottonwood/Willow, Aspen, Douglas-fir, and Krummholz. There are 581012 samples in total. In this example, 28000 samples are taken as training samples and the left are taken as test samples.

5.2.2 Analysis preparation

In this example, 5 computational nodes are used. Training data are partitioned into n sections randomly. Each section has roughly equal number data. Each attribute is normalized according to the following equation.

Let X denote attribute variable. The maximum value of X is x_{max} and the minimum value is x_{min}. The range of normalized attribute is set [0, 1]. The normalized equation is

$$x_{norm} = \frac{x - x_{\min}}{x_{max} - x_{min}}$$

5.2.3 Training process

The problem is taken as a multi-value classification problem. Multiclass classification is realized with pairwise method, i.e. k class SVM is realized through $k(k-1)/2$ binary SVMs. The "one against one" strategy, also known as "pairwise coupling", "all pairs" or "round robin", consists in constructing one SVM for each pair of classes. Thus, for a problem with k classes, k(k-1)/2 SVMs are trained to distinguish the samples of one class from the samples of another class. Usually, classification of an unknown pattern is done according to the maximum voting, where each SVM votes for one class.

In this example, C-SVC model is adopted. The parameter of the SVM model is set as follows. Constant C is set 1, radial basis function is taken as kernel function, and gamma is set as 0.01. Firstly, the example is analyzed with only 1 computation node, i.e. classical SVM method is used to train the SVM model. The trained model is used to predict the testing samples. The training time and classification correct rate are listed in Table 2. Secondly, the example is analyzed with the parallel SVM based on map/reduce. For comparison, the sample is partitioned into 2, 3, 4, 5 sub-samples respectively. When the sample is partitioned into 2 sub-samples, 2 computing nodes are used. The training time and classification rate corresponding to each partition form are listed in table 3.

Table 3 analysis results with different partition nodes

nodes Number	Number of SVs	Training time(s)	Classification correct rate
1	22177	396.125	58.76
2	21831	297.32	58.35
4	20941	251.402	58.24
8	20139	219.346	57.96

5.2.4 Feature selection with correlation coefficient

In the example, there are 54 attribute variables. Most variables provide minor contribution in the classification, especially the Soil_Type variables. For improve the training speed and reduce the noise effect of attribute variables, correlation coefficient Eq. (9) is used to select attribute variables. After calculating the correlation coefficient, the

pruning value is set 0.1. At last, 18 attribute variables are selected. The training result is listed in table 4.

Table 4 analysis result of SVM based on feature selection

Number of nodes	Number of SVs	Training time(s)	Classification correct rate
1	14624	82.022	57.46
2	14198	58.899	57.99
4	13298	55.154	58.57
8	12207	45.029	58.65

5.2.5 Result analysis

This example is a multiclass classification problem. How to improve classification correct rate of multi-class is still a big problem. From Fig.5 we can find that the training time can be reduced greatly when the sample is partitioned 2 parts. But with the increase of partition number, the training time reduction will become slow. In this example, the ratio $\alpha \approx 0.4$ and $\beta \approx 1.2$. It is similar to the analysis problem of example 1. With the introduction of feature selection, the computation can be reduced greatly. From the analysis correct rate we can find that correct rate will not decrease too much.

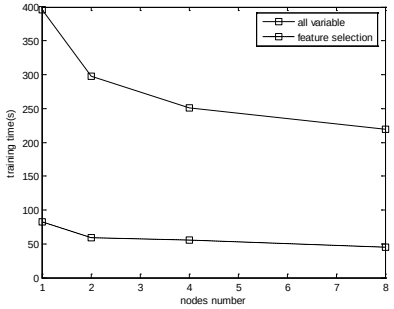


Fig. 5 training time based on different partition nodes

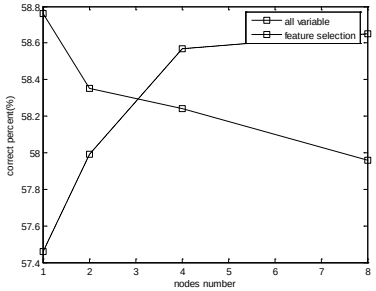


Fig. 6 Correct rate based on different partition

5.3 Heart disease classification

5.3.1 Data source

There are 270 clinic reports. Each report includes 13 factor variables. Clinic is divided into 2 classes. For testing the efficiency of the proposed cascade SVM, we replicate the data 500 times, 1000 times, and 2000 times separately. The generated data sets has 135000, 270000, 540000 samples separately. The initial data set is used to test the SVM model. The training time and correct rates based on different partition styles are listed in table 5, table 6 and table 7 respectively.

Table 5 analysis result with data replicated 500 times

Number of nodes	Number of SVs	Training time(s)	Classification correct rate
1	7585	313.755	99.629
2	7712	148.184	99.259
4	7690	87.523	98.518
8	7487	76.773	98.148

Table 6 analysis result with data replicated 1000 times

Number of nodes	Number of SVs	Training time(s)	Classification correct rate
1	8972	539.28	100
2	9055	234.49	99.63
4	8739	123.887	98.15
8	8688	86.503	97.41

Table 7 analysis result with data replicated 2000 times

Number of nodes	Number of SVs	Training time(s)	Classification correct rate
1	N/A	N/A	N/A
2	9901	578.507	100
4	9650	266.587	99.63
8	9202	158.531	99.63

The analysis result is shown as in figure 7, figure 8 and figure 9.

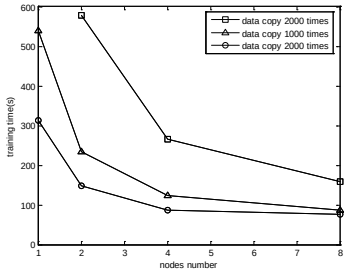


Figure 7 training time based on different partition nodes

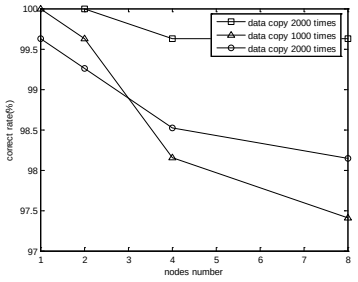


Figure 8 Correct rate based on different partition

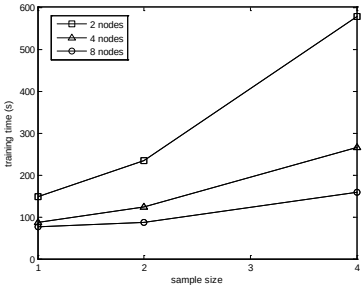


Fig. 9 training time based on different parallelism corresponding to different sample size

5.3.2 Result analysis

From the above analysis results we can find that the bigger the sample size the more obvious of the speed up.

From figure 7, we can find that when the sample size is very big, i.e. 540000 samples, it can't be processed with one single computation node. It is out of the memory. It is necessary to process big size problem with parallel style. The training time will decrease slowly when the parathion number is bigger than 8. It is because of two reasons. The first reason is that he ratio between optimum computation time and data transform overhead is less. The other reason is that the sample size of the last level can't be less than the number of support vectors. The computation cost will account a big proportion. So the computation will decrease very slowly. From fig. 9 we can find the computation time based on different partition style is approximate linear relationship to sample size.

6 Conclusions

Data-intensive data mining is still a big problems faced by computer scientist. SVM is taken as a most efficient classification and regression model. The computation cost of SVM is square proportion to the number of training data. Classical SVM model is difficult to analyze large scale practical problems. Parallel SVM can improve the computation speed greatly. In this paper, parallel SVM model based on iterative MapReduce is proposed. It is realized with Twister software. Through example analysis it shows that the proposed cascade SVM based on Twister can reduce the computation time greatly. But it doesn't mean that it is better to partition sample data into many parts. The computation time will not decrease through the analysis of the computation time. The partition number can be estimated according to the concrete problems. For increase the computation speed, the cascade SVM can be combined with other feature selection and feature extraction methods. In total, the analysis results show that the parallel SVM based on iterative MapReduce is efficient in data intensive problems.

Acknowledgements

This work is partially supported by Provincial Outstanding Research Award Fund for young scientist (No. BS2009DX016) and Provincial Fund for Nature project (No. ZR2009FM038).

References

[1] J R Swedlow, G Zanetti, C Best. "Channeling the data deluge". Nature Methods, 2011, 8: 463-465.

[2] G C Fox, X H Qiu et al. "Case Studies in Data Intensive Computing: Large Scale DNA Sequence Analysis." The Million Sequence Challenge and Biomedical Computing Technical Report, 2009

[3] X H Qiu, J Ekanayake, G C Fox et al. "Computational Methods for Large Scale DNA Data Analysis." Microsoft eScience workshop, 2009

[4] J A Blake, C J Bult. Beyond the data deluge: "Data integration and bio-ontologies." Journal of Biomedical Informatics, 2006, 39(3), 314-320.

[5] J Qiu. "Scalable Programming and Algorithms for Data Intensive Life Science." Applications Data-Intensive Sciences Workshop, 2010

[6] R Guha, K Gilbert, G C Fox, et al. "Advances in Cheminformatics Methodologies and Infrastructure to Support the Data Mining of Large, Heterogeneous Chemical Datasets." Current Computer-Aided Drug Design, 2010, 6: 50-67.

[7] C C Chang, B He, Z Zhang. "Mining semantics for large scale integration on the web: evidences, insights, and challenges." *SIGKDD Explorations*, 2004: 6(2):67-76.

[8] G C Fox, S H Bae, et al. "Parallel Data Mining from Multicore to Cloudy Grids." High Performance Computing and Grids workshop, 2008

[9] B J Zhang, Y Ruan et al. "Applying Twister to Scientific Applications." Proceedings of CloudCom, 2010

[10] J Ekanayake, H Li, et al. "Twister: A Runtime for iterative MapReduce." The First International Workshop on MapReduce and its Applications of ACM HPDC, 2010

[11] C. Cortes, V. Vapnik. "Support Vector Networks." Machine Learning,1995, 20: 273-297

[12] C C Chang, C J Lin. "LIBSVM: a library for support vector machines." ACM Transactions on Intelligent Systems and Technology, 2011, 27(2): 1-27.

[13] B Boser, I Guyon, V Vapnik. "A training algorithm for optimal margin classifiers." The 5th Annual Workshop on Computational Learning Theory, 1992.

[14] R E Fan, P H Chen, C J Lin. "Working set selection using second order information for training SVM." Journal of Machine Learning Research, 2005, 6: 1889-1918.

[15] H P Graf, E Cosatto, et al. "Parallel support vector machines: the Cascade SVM." Advances in Neural Information Processing Systems, MIT Press, 2005.

[16] A Tveit, H Engum. "Parallelization of the Incremental Proximal Support Vector Machine Classifier using a Heap-based Tree Topology." Tech. Report, IDI, NTNU, Trondheim, 2003.

[17] J X Dong, A Krzyzak, C Y Suen. "A fast Parallel Optimization for Training Support Vector Machine." Proceedings of 3rd International Conference on Machine Learning and Data Mining, 2003: 96-105.

CUERA: A generic data- and undo/redo-consistency framework for realtime interactive collaboration applications

Daniel Stolzenberg and Erika Müller
Institute of Communications Engineering, University of Rostock, Germany

Abstract— *Existing solutions for eventual consistent any-undo/redo in collaborative applications are very limited to specific domains. Thus a widely applicable framework based on entity-relationship-models is introduced. It defines generic state-focusing action types and undo/redo meta-actions. Convergence is based on a transitive precedence order and is achieved by applying a last-writer-wins scheme to an efficient scope organization of action history. Its performance scales for data and action history, but degrades regarding the number of sites.*

Keywords: CSCW, any-undo/redo, entity relationship models, eventual consistency, precedence transitivity

1. Introduction

Realtime interactive collaboration applications (RTICAs) let multiple users at different locations interact within a shared context at the same time. Since users demand high responsiveness and fluid interactivity, all data making up the shared context has to be *optimistically replicated* [1]. Consequently, the most challenging aspect of RTICAs is to ensure *eventual consistency*. That many systems developed so far are error-prone [2] or inefficient [3], highlights the severeness and importance of this consistency challenge.

Moreover, users must be able to recover from erroneous actions any user may have performed. Therefore RTICAs have to provide collaborative undo/redo features, that further complicate consistency maintenance. And since existing correct and efficient solutions for collaborative undo/redo are very limited to specific domains, consistency has to be solved independently for every non-trivial project.

To reduce complexities and efforts of this task, the generic *collaborative undoable entity-relationship-actions* (CUERA) framework is introduced. It provides a state-based solution sufficient for many RTICA projects. When better conflict resolution is needed, it can be extended to a domain specific operation-based solution. This way, developers are freed from designing a consistency solution from scratch.

But CUERA's full potential is in building the core of a data modeling, propagation and persistency platform for collaborative applications. Such an replicated model layer "out-of-the-box" would completely free developers from dealing with the intricacies of consistency maintenance: Just specify the desired data model and use derived model objects in the application's controllers and views.

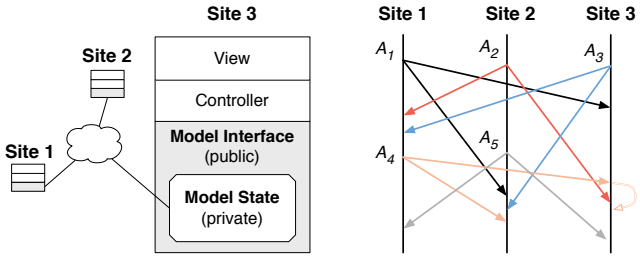


Fig. 1: Collaboration Network, Time-Space-Diagram

2. System Model and Problem Definition

2.1 Collaboration Network

The *collaboration network* is a dynamic set of *sites* S_i with equal roles. Sites can enter or leave the network at any time and any pace. Each site maintains a local *replica* of shared context data. Any local replica can be modified at any time by performing some *action* A_j. Local actions are executed immediately, assigned a unique ID_{act} and then propagated asynchronously in the network. No assumptions are made on propagation technique, time and order, but eventually all actions are received and executed at every site in the network - including late joining sites.

Each site in the network provides a graphical interface, that enables user interaction with the collaboration context. The underlying design follows the MVC pattern: The user interface elements make up the *view*, that is mediated by *controllers* to represent the current state and interaction facilities of the *model*, where replicated context data resides. Typically the view represents only a part of the model bound to the current region of interest. Moreover, the model sends *change notifications* to controllers observing this region.

In principle, the model layer is a *graph of objects*, each featuring a *private state* of encapsulated member variables and references to other objects. Based on this state, a programmatic *model interface* exposes functionality and behavior to the controllers (Figure 1). In general, all parts of this model interface, that actively update the model state, must be captured by actions propagated in the collaboration network. But since model interfaces usually provide rich and complex capabilities and are subject to regular changes, assuring consistency is a daunting task. In addition, model interfaces are highly application specific and thus cannot be generalized directly.

2.2 Consistency of RTICAs

To target consistency of user experience in RTICAs, Sun introduced the *CCI model* [4]. Its requirements depend on timings of action propagation and execution, which are commonly depicted in time-space-diagrams like Figure 1.

Causality Preservation: Users expect that semantic cause-effect-relations between actions are preserved. Since sequential timing is a necessary condition for causality, each pair of actions is determined to be either in *happened-before* ($\rightarrow$) or *concurrent* ($\|$) relation [5]. Then strict preservation of happened-before relations at all sites is sufficient to respect any potential causality. Therefore execution of an action has to be delayed, until all actions that happened-before its generation have been executed – so A_4 in Figure 1 is queued at Site 3 until A_2 from Site 2 is received and executed.

Convergence: Due to asynchronous propagation, sites can significantly differ regarding actions already executed (Figure 1) and replicas diverge temporarily. But optimistic replication assumes, that replicas converge to a consistent state, when eventually all actions are executed at all sites. But this assumption of convergence is questionable for actions that were initiated concurrently: Because they are not ordered a priori, they can and will be executed in differing sequences ($\mapsto$) (Figure 1). Since actions do *not inherently commute* in general [4], different execution orders lead to diverging states and permanent replica inconsistency. Therefore commutativity of concurrent actions has to be forced in order to achieve convergence [3].

Intention Preservation: But in order to meet intuitive expectations of users, it is not sufficient to converge to any consistent state. Rather this state has to preserve intentions of all concurrent actions [4]. The intention of an action is commonly defined as the effect of its local execution at the initiating site [4], [3]. Preserving effects of all actions is not always possible, since actions can express conflicting user intentions. As manual conflict resolution is no option in RTICAs, convergence can only be assured by coherent *automatic conflict resolution* [1]. This criterion has turned out to be quite problematic, because it is under-formalized and hard to prove [6]. Despite this criticism, intention/effect preservation has to be taken account of when designing eventually consistent RTICAs.

2.3 Undo/Redo

Supporting undo/redo is essential, even for single user applications. But research has shown that it has to be recognized as an error recovery user intention rather than a fixed system function [7]. So several undo/redo modes have been proposed, that enable users to express intentions for error recovery (Figure 2).

Meanwhile *chronological undo* has become the de facto standard, because it is powerful, yet easy to use and users can anticipate its behavior. But its simplicity also causes a major restriction: To undo an action, all subsequent actions have to be discarded as well. To overcome this limitation, *selective undo* allows users to undo/redo *any* action. But simply employing the inverse action does not always reveal the correct result for this undo mode. So achieving the correct effect is more difficult for the system and its anticipation more demanding for users.

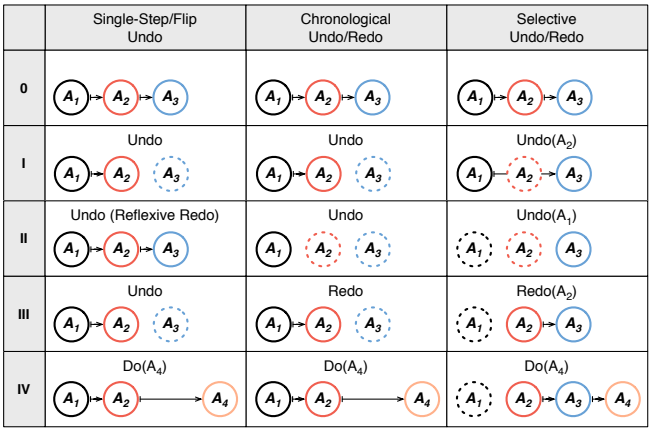


Fig. 2: Important Undo/Redo Modes

Generally, RTICAs should be compatible to features of applications, users are accustomed to [8]. But in addition, error recovery in RTICAs is of increased importance:

- Errors potentially have impact on other users' work.
- Probability of errors increases due to concurrency.
- Conflict resolution can produce unintended results.
- Fear of irreversible errors hinders explorative usage.
- Acts of vandalism are possible in open systems.

If undo/redo provides a powerful way to recover from errors caused by collaboration, it can help users managing its complexity [9]. But supporting collaborative undo/redo in a compatible way is more complex. Fortunately, undo/redo concerns can be separated into different components [8]:

A *history* component stores all actions performed in the network, including initiating site and local execution order.

The *selection* component keeps track of all information, that is needed to identify, which actions are intended to be undone/redone. The user facing details of selection depend on the current state, mode and scope (local/global) of undo/redo and are out of the focus of this paper.

The *algorithm* component has to assure, that replicated data consistently reflects the actions as specified by their undo state. Undo/redo itself underlies concurrency as well and therefore has to converge to achieve consistency. As undoing and redoing the same action is non-commutative per se, convergence control beyond actual data is needed. And even plain data convergence is influenced by undo/redo: Because concurrent actions of all users are interleaved and executed in different orders, actions may be located very differently in the history of each site. Still undoing/redoing an action has to reveal consistent data.

3. Related Work

Based on the previous section, relevant research related to the scope of this paper can be identified. Most of this is rooted in the field of collaborative text editing systems, featuring $insert$ and $delete$ operations on a linear data type. *Operational Transformation* (OT) [10] dominates research in this area and has been extended to a wide range of domains [11], [12]: Conceptually, concurrent operations are transformed against each other to let different execution orders converge. In fact, elaborating this basic idea has led to a plethora of algorithms [13], most of them failing in special situations called "puzzles" [14], [6]. Algorithms known to be correct are very complex to design in practice and computationally expensive when implemented [15], [16].

In reaction, some recent research targets convergence by commutativity of operations without transformation. In particular, Shapiro et al. [2] and Roh et al. [3] developed commutatively replicated variants of important abstract data types (Counters, Registers, Sets, Arrays and even Graphs). But none of them offers the spectrum of capabilities targeted in this paper and especially undo/redo has been neglected.

Though many OT based systems support undo/redo of any action [17][18], non-OT research on this matter is rare. Most of it is related to text editing in P2P-Wikis [19] [20]: Besides the limitation to sequential data, their counter-based undo/redo results in a user experience not suitable for realtime collaboration. The latter also holds for the XML tree editing system in [9], but XML nodes and attributes at least partly resemble the model object graph concepts identified in Section 2. Hence, some of its approaches can inspire the development of the intended CUERA framework: The history organization of operations in their respective scope and the LWW technique for attribute convergence. But most aspects of a generic data and undo/redo consistency solution are still unresolved and motivate this paper.

4. CUERA Framework

4.1 Data & Action Representation

One of the most promising and generic abstractions of data has not yet been directly considered in RTICA research. Entity-Relationship Models (ERMs) [21] formalize a domain of interest for a given project. Its constituting elements are classified into several *entity-types* (T_{ent}). Relevant properties of these elements are identified to become *attribute-types* (T_{att}). Similarly connections and dependencies of elements are represented as *relationship-types* (T_{rel}) between their corresponding entity-types, where *cardinalities* and *labels* for both directions are attached. Advanced features of ERMs are beyond the scope of this paper, but will be examined in future investigations: *Specialization/generalization* of entity-types, *aggregation/composition* of entities and inherently *ordered* relationships.

ERMs are the de facto standard in data modeling [22] and the vast majority of software projects start with some form of ERM to derive database schemas and class structures. Therefore ERMs can be assumed to be a widely applicable abstraction of common data needs. Moreover, concepts of ERMs match those of model object graphs (Section 2). Consequently, ERMs serve as the conceptual foundation for the intended collaborative data framework.

So a *replicated entity relationship graph* (RERG) data type is introduced, literally realizing an ERM: Its vertices and edges are incarnations of entity- and relationship-types. Actions on this data type are composed of an ordered set of *operations*, in order to capture user intentions involving more than one of the defined *operation-types* (Figure 3):

- $EntityOperation\ (T_{ent}, ID_{ent})$
  - $CreateEntity$
  - $DestroyEntity$
- $AttributeOperation\ (T_{att}, ID_{ent}, [Parameters])$
  - $SetAttribute\ (\ldots, Value)$
- $RelationshipOperation\ (T_{rel}, ID_{src}, ID_{dst})$
  - $AddRelationship$
  - $RemoveRelationship$

Entities are identified by unique and unambiguous ID_{ent} generated at the initiating site. In addition, entities are *non-reviving*: Once destroyed they cannot be created again. Relationships are *bidirectional* per se and unambiguousness is assured by a unique *source-destination-direction* for every type. So relationships and attributes do not need dedicated IDs, since they are identified by type and entity IDs.

In order to ensure integrity of this graph, relationships using destroyed entities are considered *invalid*. Moreover, ERM conformance implies another constraint: An entity can only be connected to a single entity for a relationship-type with singular cardinality in this direction ("To-One"). Hence such relationships are considered to be *replaced* by new relationships involving the same type and entity.

In RTICAs, these constraints have to be obeyed in face of concurrent changes. This is simplified by the RERG approach, because relationships are represented explicitly in comparison to the implicit reference representation normally used in model object graphs. Nevertheless, the familiar reference representation can be translated to and from the explicit relationship representation.

This representation of data and actions focuses on the private state underlying the model interface to the controller layer. So of any kind of interaction, only the resulting changes to the state of entities, attributes and relationships are captured. This state-based approach is the enabling factor for the generic abstraction of rich, complex and application specific model interfaces. Particularly, the amount of action-types is drastically reduced and consistency maintenance is simplified. But of course, this state-based replication also has disadvantages [1], that will be discussed in Section 5.

4.2 Convergence Control

Section 2.2 demonstrated that assuring CCI consistency is dependent on commutativity of concurrent actions. So all combinations of operation-types are analyzed for inherent commutativity in a naive implementation, that respects the aforementioned relationship invalidation and replacement. It is obvious, that concurrent actions are commutative as long as affected entities and relationships do not overlap. Within the same region of the graph, some combinations can never occur: Due to unique and non-reviving entities, strictly no action can involve an entity created concurrently. Only three of the remaining combinations do not commute inherently:

- Concurrently setting the same attribute of the same entity to different values does not commute, since the attribute always reflects the last executed operation.
- Concurrently adding and removing the same relationship does not commute, because the relationship always reflects the last executed operation.
- Concurrently adding different relationships to the same singular cardinality does not commute, because the last executed addition replaces all previous ones.

Convergence of concurrent actions, that do not commute, must be forced. This requires both a conceptual framework and a solution specific to the RERG data type. This paper makes use of the *Operation Commutativity by Precedence Transitivity* (OCbyPT) framework by Roh et al. [3], because it provides a useful guideline for the design and proof of eventually consistent systems. Conceptually, it establishes an artificial total *precedence order* of actions, that respects happened-before relations. The convergent state is then defined as the result of naively executing actions in this precedence order, thereby automatically resolving intention conflicts. Any other valid causal execution order has to reveal exactly this state by commuting actions, *as if* they were executed in precedence order. This can be achieved by keeping some metadata: Based on the knowledge of already executed actions and their precedence relations (PR) to a received action, its correct execution effect leading towards the consistent state can be inferred. Although this approach resembles serialization, *it does not restrict action execution to precedence order*.

When OCbyPT is applied to the RERG data type, it allows to utilize the simple Thomas‘ write rule (LWW - Last Writer Wins) to force convergence and resolve intention conflicts. Because of the state-based approach, only one operation is *effective* for every RERG element:

- The set-attribute with the highest precedence is effective for an attribute and defines its value.
- The addition or removal with the highest precedence is effective for a relationship and defines its existence.
- The relationship with the highest addition precedence is effective for a singular cardinality – regardless of being removed, invalidated or replaced (in "One-To-One").

To preserve semantic cause-effect-relations, precedence relations ($\rightsquigarrow$) must include and generalize happened-before relations ($\rightarrow$). So in practice, they are derived from the same distributed *clock* mechanisms needed for causality preservation anyway [3]. Thus, in addition to expressing happened-before relations, *timestamps* unambiguously order and identify actions by precedence. Although no particular technique is assumed in this paper, disadvantages related to distributed clocks in general are discussed in Section 5.

4.3 Undo/Redo Representation

The convergence control based on OCbyPT and LWW already implies the fundamental approach to undo/redo: The correct effect is defined by the result of executing only actions in precedence order, that are *active* (i.e. not undone). So it must be defined, how this state of actions is altered.

Since a closed action expressing a user intention generally involves several RERG operations, undo/redo must reflect this by restricting granularity to the action level. In addition, the same holds for undo/redo itself: Expressing an error recovery user intention generally requires to undo/redo more than one original action. And because performing undo/redo does not directly alter the RERG state, but rather the state of actions, undo/redo is represented by *meta-actions* composed of an ordered set of *meta-operations* (Figure 3):

$MetaOperation(ID_{act})$
- $UndoAction$
- $RedoAction$

Affected actions are identified by their IDs and if only original actions are targeted, undo/redo is non-reflexive: To undo an undo meta-operation, the original action has to be redone. This allows to hide the existence of meta-actions, which simplifies user experience to the state of original actions. Concurrently undoing and redoing the same action does not commute, so undo/redo convergence has to be forced as well. Thus meta-operations take part in the same LWW scheme used in RERG operations: The meta-operation with the highest precedence is effective for an action and defines its *meta-state*. Because of the many commonalities, actions and meta-actions are abstracted by *events*.

4.4 Model State Retrieval

To enable efficient model retrieval, the CUERA framework exactly reflects LWW convergence and undo/redo meta-state control at runtime. Each element references its *affecting* (meta-)operations sorted by precedence. The one with the highest precedence and active meta-state is tracked as its *effective* (meta-)operation. This *scope organization* is depicted in the *timestamp event graph* (TEG) in Figure 3.

To capture replacement at any singular cardinality, relationships additionally track their topmost preceding $AddRelationship$ operation. This allows entities to track the effective relationship at singular cardinalities. Model state retrieval is carried out based on this scope organization:

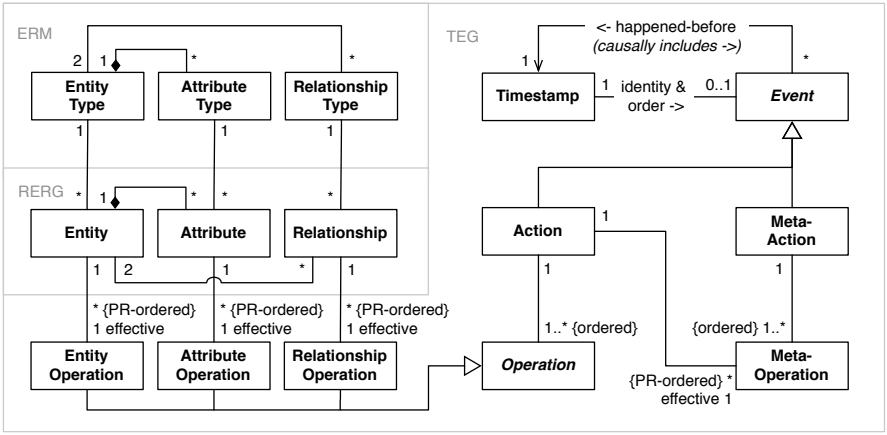


Fig. 3: CUERA Framework

- For an entity to exist, its effective operation must be of $CreateEntity$ type.
- If an attribute's surrounding entity exists, its state is defined by the effective operation's $Value$ parameter.
- To provide a familiar model interface, relationships have to be translated into a reference representation: If the entity at the origin of the retrieval exists, connected relationships of the type in question are inspected (for a singular cardinality at the origin of retrieval only the effective relationship is considered):
  - The existence of the relationship itself is checked: Its effective operation must be of $AddRelationship$ type.
  - The target entity is tested for existence.
  - In case of singular target cardinality, it is examined if the relationship is the effective one.
  - If all conditions are met, a reference to the target entity is included in the result.
- An action is only active regarding meta-state, if no $UndoAction$ operation is in effect.

4.5 Local & Remote Event Execution

Local interaction with the public model interface has to be recorded into a (meta-)action object. To assure that a local event always precedes all existing events, first the timestamps are attached to the event with the clock being incremented in between. Then the event components can be recorded:

- $Create/DestroyEntity$ operations for entities. Moreover, a fresh ID_{ent} is generated for created entities.
- $SetAttribute$ operations for attributes.
- Again, the reference representation has to be translated into relationships: From the ID_{ent} of each reference the $Add/RemoveRelationship$ operation can be build. In addition, the relationship element in the RERG has to be created if necessary.
- $Undo/RedoAction$ meta-operation for actions.

To be reflected in the private model state, each recorded (meta-)operation must be inserted into its affected elements' scope. Because of its most recent timestamp, it immediately becomes the effective operation, which in turn triggers a change notification: For example, when an $UndoAction$ meta-operation becomes effective on its affected action, the action changes to inactive meta-state. So it notifies all its contained operations, which notify their respective affected elements of becoming inactive. This process goes down the chain of dependent elements, updating the scope of each affected element and notifying all model observing controllers in the end.

Finally, the local event is encoded and propagated in the collaboration network. After reception at a remote site, the event is decoded before it can be executed. But causality constrains the execution of remote events: If happened-before relations were violated, the execution must be delayed until all these events are received and executed (Section 2.2).

When it is causally ready for execution, the remote event is integrated in the private model state representation. At first, affected elements must be retrieved or created, if they do not exist yet. Then, the same as for local events, each (meta-)operation is inserted into the scope of its affected element: Precedence of the received (meta-)operation is compared to all existing ones in this scope in top-down progression. If it becomes effective and the resulting state changes, a change notification is triggered.

4.6 Undo/Redo Selection

Any framework has to support various undo/redo selection modes, that depend on a total order of all executed actions. Thus the *perceived causal execution sequence* (pCES) is provided, that mirrors the unique history of every site by appending actions upon execution. A selection order equal for all sites is the *precedence ordered sequence* (POS), that is particular useful, since it exactly reflects convergence control and thus eases anticipation of the undo/redo effect for users.

5. Evaluation

5.1 Consistency Correctness

The CUERA framework preserves causality by delaying execution of remote events that violate happened-before relations. Convergence can be divided into two conceptual layers. The RERG and TEG representations are inherently commutative: Elements are only added and element orders are based on the globally consistent precedence definition. Thus RERG and TEG are eventually consistent. Finally, the model state is retrieved and converges by applying LWW rules theoretically backed by the OCbyPT framework.

While CUERA preserves causality and convergence, its intention preservation capabilities are limited for conceptual reasons. In general, any generic data consistency solution has to abstract application specific model interfaces. The CUERA framework accomplishes this by focusing on the model state: Syntactical results of interactions are captured, while their semantical meaning is lost. So state convergence can be achieved, whereas the full potential of conflict resolution and intention preservation cannot be tapped [1].

For example, imagine two concurrent interactions that both result in incrementing the same "counter" attribute. An application-specific solution could increment the counter twice, but the state-based RERG ignores one incrementation, since both are interpreted as setting the counter to the same value. While in this example all intentions can be preserved in theory, this is not always possible. Another interaction, e.g. resetting the counter to zero, will inherently contradict concurrent incrementations as well.

While some intentions are inherently contradicting and cannot be preserved, still a significant fraction of intentions violated by the generic RERG could be preserved by specific data types. To which extent such conflicts are caused in practice, is a really complex matter dependent on many different factors (like concurrency interval, interaction rate, number of participants, group focus, application domain, model interface, ERM design ...). If their overall probability and frequency as well as their percentage increase beyond a certain degree, the user experience is affected perceivably and becomes ineffective and distracting.

Therefore the generic state-focusing operation-types of this framework can be insufficient. In the "counter" example above, an alternative ERM design is possible: Both the counter itself and its incrementations can be refactored into connected entity-types. Using these, a "virtual" counter attribute can be derived for the public model interface.

In addition, CUERA can be enhanced with application specific operation-types. In this case, $CounterAttribute$ can be added, that accumulates effective $IncrementCounter$ operations with higher precedence than the effective $SetAttribute$. Hence CUERA still provides a valuable framework, even when the need for better conflict resolution arises.

5.2 Time Complexity

RTICA performance is dependent on time complexity regarding factors that underlie scaling. Thus model retrieval and event execution in the CUERA framework must be analyzed regarding the number of:

- sites in the network s
- events in the history $|H|$
- data elements N

Entities and actions can be retrieved in constant time using ID maps. The same holds for attributes, since they are linked to their surrounding entity. Retrieval of references involves translation of multiple relationships, entities and replacements: Still their number is independent from the above factors in practice, since only connected relationships of the type in question and their target entities are inspected. Because each element tracks its effective (meta-)operation, state retrieval performs with constant time complexity $O(1)$.

Upon execution, both local and remote events and their components are integrated in the RERG and TEG. This process involves several tasks: Element retrieval, creation and registration all perform in $O(1)$. Then (meta-)operations are inserted in the affected elements‘ scope. This is carried out in top down progression, because final locations of operations are likely to be near the top. Hence, insertion only involves precedence comparisons to operations with the same scope and higher precedence. Therefore this subset is denoted as $|h_{\rightsquigarrow}|$. In particular, it is independent from and significantly smaller than the total number of events in the history. But the amount of operations already present and of higher precedence depends on the number of sites, so $|h_{\rightsquigarrow}| \sim s$. Fortunately, distributed clock techniques allow precedence comparisons in constant time after initial computation. So insertion performs with $O(s)$.

If becoming effective, event components trigger change notification down the chain of affected elements, which is of constant size. To reflect the inserted event, each affected element has to update its current effective operation and state. This involves a number of precedence comparisons to operations in its scope, which is independent of the factors in question. So change notification performs in $O(1)$.

In summary, CUERA performs in linear time in regard to network size $O(s)$ due to precedence comparison in scope organization. As a consequence, scalability is limited in terms of participating sites. But on the other hand, performance scales perfectly in respect to data and event history, which is the benefit of operation scope organization.

5.3 Space Complexity

The CUERA framework features linear space complexity regarding events and data (including obsolete elements). In addition, events maintain timestamps and although there are efficient distributed clock mechanisms, space consumption increases with the number of sites. Thus space complexity is $O(N + s \times |H|)$.

Obviously, the space overhead in respect to the current model state is huge: Not only obsolete events in the TEG, but also unnecessary elements in the RERG must be kept to support full undo/redo of any action. But the undo/redo scope is limited in practice, since the probability of a user undoing/redoing an action rapidly decreases with its age. Furthermore, undo/redo in single user applications does not stretch beyond persisted states. How these fact can be exploited to purge the RERG and TEG is left open for future research.

6. Conclusion

To ease the development of collaborative applications, this paper has introduced the CUERA framework - a generic solution to data and undo/redo consistency. It is applicable in a wide range of domains, because data is represented in a literal realization of an entity relationship model (RERG). To provide a generic consistency solution, a set of state-focusing operation-types has been defined. In addition, non-reflexive meta-operations allow to undo/redo any action. To ensure convergence, a LWW scheme based on transitive precedence relations is employed (OCbyPT). Efficient state retrieval, event execution and notification of changes are facilitated by the precedence ordered scope organization of event components (TEG). Beyond undo selection in local execution order (pCES), the global precedence order (POS) is provided, that exactly mirrors convergence control and thus eases undo/redo effect anticipation for users.

Correctness of the CUERA framework according to CCI has been evaluated and two optimization strategies have been proposed to overcome the conceptual weakness of the generic solution regarding intention preservation. And furthermore, performance has been examined: All processes perform in $O(s)$ or better, therefore the framework does not scale well in terms of network size. But on the other hand it scales perfectly in regard to data and event history.

In summary, the CUERA framework provides a generic and efficient consistency solution for RTICAs: Developers are freed from the most complex intricacies of collaborative applications. But only if the framework is employed as the core of a distributed data platform, its full potential is unleashed: By simply specifying the desired ERM, developers can be provided with an "out-of-the-box" data layer. So the CUERA framework can potentially lead to a proliferation of realtime interactive collaboration applications.

In future research, the missing ERM features will be added to tap the full potential of this approach. Moreover, some performance experiments will be conducted to prove the scalability and efficiency claims of this paper. Finally, removal of obsolete history and data elements will be examined to reduce space overhead of unlimited undo/redo.

References

[1] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys (CSUR)*, 2005.

[2] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," INRIA Rocquencourt, France, Tech. Rep. 7506, 2011.

[3] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, 2011.

[4] C. Sun, Y. Zhang, X. Jia, and Y. Yang, "A generic operation transformation scheme for consistency maintenance in real-time cooperative editing systems," *GROUP '97: Proceedings of the international ACM SIGGROUP conference on supporting group work*, 1997.

[5] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, 1978.

[6] R. Li and D. Li, "Commutativity-based concurrency control in groupware," *Proceedings of the First IEEE Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2006.

[7] G. D. Abowd and A. J. Dix, "Giving undo attention," *Interacting with Computers 4*, 1992.

[8] C. Sun, "Undo any operation at any time in group editors," *Proceedings of the 2000 ACM conference on computer supported cooperative work*, 2000.

[9] S. Martin, P. Urso, and S. Weiss, "Scalable xml collaborative editing with undo," *OTM 2010*, 2010.

[10] C. Ellis and S. Gibbs, "Concurrency control in groupware systems," *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, 1989.

[11] Agustina, F. Liu, S. Xia, H. Shen, and C. Sun, "Comaya: Incorporating advanced collaboration capabilities into 3d digital media design tools," *CSCW '08: Proceedings of the 2008 ACM conference on Computer supported cooperative work*, 2008.

[12] C. Palmer and G. Cormack, "Operation transforms for a distributed shared spreadsheet," *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, 1998.

[13] S. Kumwat and A. Kkunteta, "A survey on operational transformation algorithms: Challenges, issues and achievements," *International Journal of Computer Applications*, 2010.

[14] A. Imine, P. Molli, G. Oster, and M. Rusinowitch, "Proving correctness of transformation functions in real-time groupware," *ECSCW'03: Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work*, 2003.

[15] G. Oster, P. Urso, P. Molli, and A. Imine, "Real time group editors without operational transformation," INRIA Rocquencourt, France, Tech. Rep. 5580, 2005.

[16] N. Preguiça, J. Marques, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," *ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, 2009.

[17] B. Shao, D. Li, and N. Gu, "An algorithm for selective undo of any operation in collaborative applications," *GROUP 2010*, 2010.

[18] S. Weiss, P. Urso, and S. Molli, "An undo framework for p2p collaborative editing," *CollaborateCom 2008*, 2009.

[19] C. Rahhal, S. Weiss, H. Skaf-Molli, P. Urso, and P. Molli, "Undo in peer-to-peer semantic wikis," *ESWC 2009*, 2009.

[20] S. Weiss, P. Urso, and S. Molli, "Logoot-undo: Distributed collaborative editing system on p2p networks," *IEEE transactions on parallel and distributed systems*, 2010.

[21] P. P.-S. Chen, "The entity-relationship model - toward a unified view of data," *ACM Transactions on Database Systems (TODS)*, 1976.

[22] ——, "Entity-relationship modeling: Historical events, future trends, and lessons learned," *Software Pioneers: Contributions to Software Engineering*, 2002.

Increasing the Efficiency of Distributed Goal-Filling Algorithms for Self-Reconfigurable Hexagonal Metamorphic Robots

J. Bateau[1], A. Clark[1], K. McEachern[1], E. Schutze[1], and J. Walter[1]
[1]Computer Science Department, Vassar College, Poughkeepsie, NY, USA

Abstract— *The problem addressed is the distributed reconfiguration of a system of two-dimensional, hexagonal mobile robots (modules), from an initial straight chain into an arbitrary shaped, connected goal configuration that satisfies a simple admissibility condition.*

We present algorithms that improve the efficiency of the deterministic algorithms presented in [16] and [15]. In this paper, we first present a new algorithm for optimally filling a chain of cells that bisect the goal configuration. Then we present reconfiguration algorithms that combine techniques used in the goal-filling algorithm papers cited above combined with the bridging algorithms presented in [7] and [4]. We compare the performance of our new algorithms to existing goal-filling algorithms via simulation using a discrete event simulator. The results of our simulation are presented and discussed.

Keywords: Metamorphic robots, distributed reconfiguration, self-reconfiguration

1. Introduction

A *self-reconfigurable* robotic system [5] is a collection of independently controlled, mobile robots, each of which has the ability to connect, disconnect, and move around adjacent robots. *Metamorphic* robotic systems [3] are a subset of self-reconfigurable systems. In metamorphic systems, each module is identical in structure, motion constraints, and computing capabilities. The modules have a regular symmetry so that they can be assembled with no gaps between adjacent modules. In metamorphic robotic systems, robots achieve locomotion by moving over a substrate composed of one or more other robots. The mechanics of locomotion depend on the hardware and can include module deformation to crawl over neighboring modules [3], [13] or to expand and contract to slide over neighbors [14]. Alternatively, moving robots may be constrained to rigidly maintain their original shape, requiring them to roll over neighboring robots [10], [19], [20].

Shape changing in these composite systems is envisioned as a means to accomplish various tasks, such as bridge building, structural support, satellite recovery, tumor excision [13], and automated movement of two-dimensional arrays of solar collectors or shields. The complete interchangeability of the robots provides a high degree of system fault tolerance. Self-reconfiguring robotic systems may be potentially useful in environments that are not amenable to direct human observation and control (e.g., interplanetary space, undersea depths) or for tasks that are monotonous for humans.

The motion planning problem for a metamorphic robotic system is to determine a sequence of robot motions required to go from a given initial configuration (I) to a desired goal configuration (G).

Many existing motion planning strategies rely on centralized algorithms to plan and supervise the motion of the system components [3], [5], [13], [14], [18]. Others use totally distributed approaches which rely on heuristic approximations or require communication between robots in each step of the reconfiguration process [2], [10], [11], [19], [20].

We focus on a system composed of planar, hexagonal robotic modules as described by Chirikjian [3]. We present a motion planning strategy that assumes knowledge of all initial coordinates of cells in G. We start with a centralized phase in which modules match themselves to a position in G. This is followed by a distributed phase when as many robots as possible move in parallel. Our distributed approach offers the benefits of localized decision making, the potential for greater system fault tolerance, and less communication between modules than other approaches. We have previously applied this approach to the problem of reconfiguring a straight chain to an intersecting straight chain [17] and a straight chain to a goal configuration that satisfies a general "admissibility" condition [16], [15]. We modify the admissibility requirement on G in this paper.

2. Related work

Chirikjian [3] and Pamecha [13] discuss centralized algorithms for planar hexagonal modules that use the distance between all modules in I and the coordinates of each goal position to accomplish the reconfiguration of the system, moving a single module in each time step. Pamecha et al. [13] define the distance between configurations as a metric and apply this metric to system self-reconfiguration using a simulated annealing technique to drive the process towards completion. In [6], motion planning time is shown to be in $O(n)$ for n modules when particular motion

constraints are used on hexagonal metamorphic robots. They do not address the reconfiguration time as we do in this paper.

Centralized motion planning strategies for systems of two dimensional robotic modules are also examined by Nguyen et al. [12] and analysis is presented for the number of moves necessary for specific reconfigurations. A centralized motion planning strategy for three dimensional cubic robots is presented by Rus and Vona [14]. A set of distributed motion planning algorithms for a system of cubic robots is presented by Butler et al. in [2]. In another paper [1], Butler et al. present a rule set that can be run by vertical "layers" of cubic modules and a distributed control algorithm for locomotion is described that will work in any system composed of cubic modules.

Distributed approaches are taken by Murata, et al. to reconfigure a system of two dimensional hexagonal modules [10], and a system of three dimensional cubic modules [11]. Yim et al. [19] and Zhang et al. [20] present distributed algorithms to reconfigure three dimensional rhombic dodecahedral modules. In [9], Miao et al. present algorithms to reconfigure two dimensional hexagonal modules to envelop obstacles. Unlike the planning algorithms presented in this paper, these algorithms are probabilistic and require message passing between neighboring modules.

2.1 Our approach

This paper examines distributed motion planning strategies for a planar metamorphic robotic system undergoing a reconfiguration from a straight chain to a goal configuration satisfying certain simple properties. In our algorithms, robots are identical but act as independent agents making decisions based on their current position and the sensory data obtained from physical contacts with adjacent robots. We have shown that collision-free reconfiguration in certain scenarios, like those presented in our earlier papers [17], [16], [15], can be accomplished using algorithms that do not require any message passing. Our long term goal is to seek an understanding of the necessary building blocks for reconfiguration, starting with algorithms in which no algorithm messages need to be passed between participating robots during reconfiguration. Therefore, our algorithms are more communication efficient than the distributed approaches of [2], [10], [19] and [20].

Our proposed scheme uses a classification of robot types based on connected edges similar to the classification used by Murata et al. [10] for connected vertices. In the algorithms presented in this paper, each robot independently determines whether it is in a movable state based on the cell it occupies in the plane, the locations of cells in the goal configuration, and which of its sides are adjacent to occupied cells. Robots move from cell to cell and modify their states as they change position. Since the robots know the coordinates of the goal cells, we show that each of them can independently choose a motion plan that avoids module collision.

One of the contributions of this paper is the presentation of algorithms that allow modules to move with only a single space between them while also ensuring that moving modules do not come into contact in acute angle corners. To accomplish this, we use ideas developed in our earlier work on bridging algorithms [7], [4]. In these algorithms, certain modules temporarily halt during reconfiguration, forming bridges for other modules to cross. After all modules have passed over a bridge module, the bridge module resumes motion. We use a similar technique to avoid module collision in this paper.

3. System Model

The plane is partitioned into equal-sized hexagonal cells and labeled using the same coordinate system as described by Chirikjian [3].

Our model provides an abstraction of the hardware features and the interface between the hardware and the application layer.

3.1 Assumptions about modules

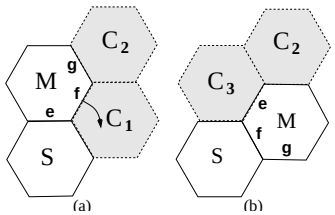


Fig. 1: Before (a) and after (b) module movement: M is moving module, S is substrate, and shaded cells are unoccupied.

- Each module is identical in computing capability and runs the same program.
- Each module is a hexagon of the same size as the cells of the plane and always occupies exactly one of the cells.
- Each module knows at all times:
  - its location (the coordinates of the cell that it currently occupies),
  - the location of the cells in G,
  - its orientation (which edge is facing in which direction), and
  - which of its neighboring cells is occupied by another module.

  Modules move according to the following rules:
  1) Modules move in lockstep rounds.
  2) In a round, a module M is capable of moving to an adjacent cell, C_1, iff (see Fig. 1 for an example)
     (a) cell C_1 is currently empty,

(b) module M has a neighbor S (called the *substrate*) that is also adjacent to cell C_1 and does not move in the round, and
(c) the neighboring cell to M on the other side of C_1 from S, C_2, is empty.

3) Only one module tries to move into a particular cell in each round.

Note that the modules may be deformable, in which case each module moves by changing joint angles to crawl over an unmoving substrate. Alternately, the modules may be rigid, using sliding movements as specified in [9] to move over the substrate.

If the algorithm does not ensure that each moving module has an immobile substrate, as specified in rule 2(b), then the results of the round are unpredictable and can lead to deadlock. Likewise, collision may result if the algorithm does not ensure rule 3.

4. Centralized Pre-processing Phase

Our objective is to design a deterministic distributed algorithm that will cause the modules to follow a collision-free plan from an initial straight chain configuration, I, to an admissible goal configuration, G. This algorithm should ensure that modules do not collide with each other, and the reconfiguration should be accomplished in the most efficient way possible (in terms of time of reconfiguration).

We assume G is oriented such that cells have flat surfaces facing north (N) and south (S). This way we can unambiguously describe the eastmost, westmost, and inner columns of G.

Definition 1: An *admissible goal configuration* is connected and has no vertical gaps within columns.

For simplicity, we require that the straight chain I consists of n modules and that it initially intersects G in one cell. The module in the cell that overlaps G does not move during reconfiguration.

In a centralized pre-processing phase, modules determine their positions in straight chain I based on their distance from the cell in I that overlaps G. Module numbering proceeds from 1 (at the greatest distance from the overlapping cell) to $n-1$ (the cell adjacent to the overlapping cell). At the start of the reconfiguration, there are $n-1$ empty cells in G.

Our previous strategy, presented in [16], to fill G while avoiding collision, was to find a contiguous path (what we call the *substrate path*, or SP) of goal cells that most evenly bisects G. After the SP cells are filled, the cells to the N and S of this path are filled in parallel, with minimal intermodule spacing. Filling the SP first guarantees that no module on the N will collide with a module on the S, allowing both segments to be filled in parallel. We designed our algorithms to further ensure that no pair of moving modules becomes adjacent throughout the reconfiguration, and for that reason, our earlier algorithms required two unoccupied spaces between each pair of modules traversing the same surface. This spacing was also applied to the modules filling the SP. Figure 2 shows the result of 2-cell versus 1-cell separation when modules move through an acute angle.

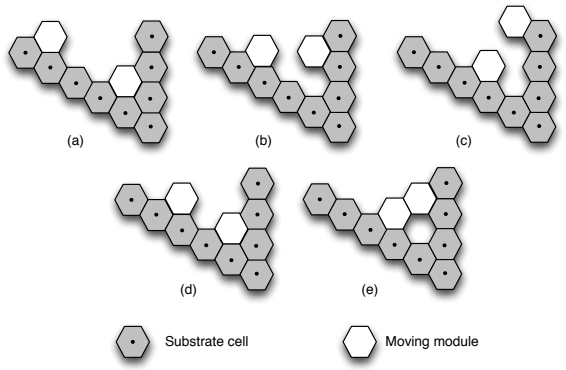


Fig. 2: Modules rotating clockwise over a substrate surface. Parts (a)-(c) show the outcome for modules with 2-cell separation. Parts (d) and (e) show how moving modules come in contact in acute angle corners if there is only 1-cell separation. In part (e), the modules are in a configuration in which the choice of substrate could jeopardize the reconfiguration if a moving module chooses another moving module as a substrate (we call this a deadlock situation).

In this paper, we place a restriction on the SP so that modules can move with a single unoccupied cell separation in both the clockwise (CW) and counter clockwise (CCW) directions.

Definition 2: An *admissible SP* is contiguous and has no vertical sections.

Our procedure for finding a SP in an admissible goal configuration is given below:

1. Find the midpoint cell in each column of the goal configuration and assign that cell to be a SP cell. If a column length is even, choose the path cell north or south (based on the distribution of cells in the next column) of where the midpoint would be if the column length was odd.
2. Check if SP found in step 1 is contiguous. If not, use algorithms presented in [16] that test every possible path in G from left to right. If any SP with no vertical segments is found, continue with reconfiguration algorithms given in this paper.
3. If the only SP found contains vertical segments, use reconfiguration algorithms described in [16] to complete reconfiguration. These algorithms use a more conservative 2 cell inter-module spacing to avoid collision and deadlock.

Since we are assuming that G is oriented in columns with cell sides normal to N and S, the slope of I (i.e., the direction from module 1 to the cell in I that overlaps

G) must be either NE or SE. If the SP has no vertical segments, there is an algorithm to fill the path using single cell spacing between each pair of moving modules. The FILLSUBSTRATEPATH algorithm is given in Fig. 4.

With one exception, to be discussed in Sect. 5, all modules in I begin moving in the round they become free, based on the cell in G to which they have been mapped. A free module is one that has one of the FREE contact patterns shown in Fig. 3. It can be seen from this figure that a FREE contact pattern is one in which a module i has at least 2 sides that are not occupied by another module and in which module i's movement (in i's local view) will not cause the system to become disconnected.

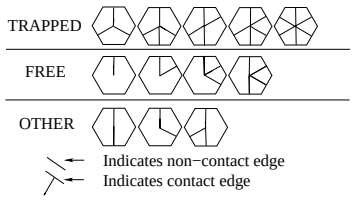


Fig. 3: Contact patterns possible in algorithm.

From Fig. 5(a) we can see that initially, the only free module in I is in position 1, furthest from G. Modules become free in order of increasing position number, with module i becoming free after the second rotation of module $i-1$.

Modules running the FILLSUBSTRATEPATH algorithm in positions i and $i+1$ (where $i \geq 1$ and $i+1 <$ length of SP) will be in the same N-S column as they move toward G. When modules move with this orientation and spacing, they are guaranteed not to collide in a contiguous SP that contains no vertical sections and that spans all columns of G. To see why, consider that exactly one of the N or S modules, x, in a moving pair of modules will always be closer, in terms of grid distance, to a particular SP cell because of the restriction on either NE or SE SP slope, and the closer modules x will therefore reach the path first.

Variables used in algorithm FILLSUBSTRATEPATH:

- *num*: Length of SP, not counting initial overlapping cell.
- *initialSlope*: Slope of I, SW to NE (called *NE* in algorithm) or NW to SE (called *SE* in algorithm).
- *lastSlope*: Slope of last two cells on SP (NE or SE).

As shown in Fig. 5, in most cases the first *num* modules fill the SP (although not always sequentially, as explained below). However, when *num* is odd and *initialSlope* $\neq$ *lastSlope*, the module in position *num*+1 reaches the SP before the module in position *num*.

Once the SP cells are calculated, the remainder of the modules are matched to positions on the N and S of the SP. Figure 5 shows the order the modules fill the SP when the length of the path is odd and the *initialSlope* $\neq$ *lastSlope*.

```
Algorithm FILLSUBSTRATEPATH()
 1. if (initialSlope == NE):
 2.       module 1 rotates CW
 3. else: // initialSlope == SE
 4.       module 1 rotates CCW
 5. if (num is even) or (lastSlope == initialSlope):
 6.       modules 2. . .num alternate directions, starting in
            direction opposite of module 1 with no delay after
            the round in which they become FREE.
 7. else: // (num is odd) and (lastSlope != initialSlope)
 8.       modules 2. . .num+1 alternate directions, starting in
            direction opposite of module 1 with no delay after
            the round in which they become FREE.
```

Fig. 4: Algorithm for modules on substrate path.

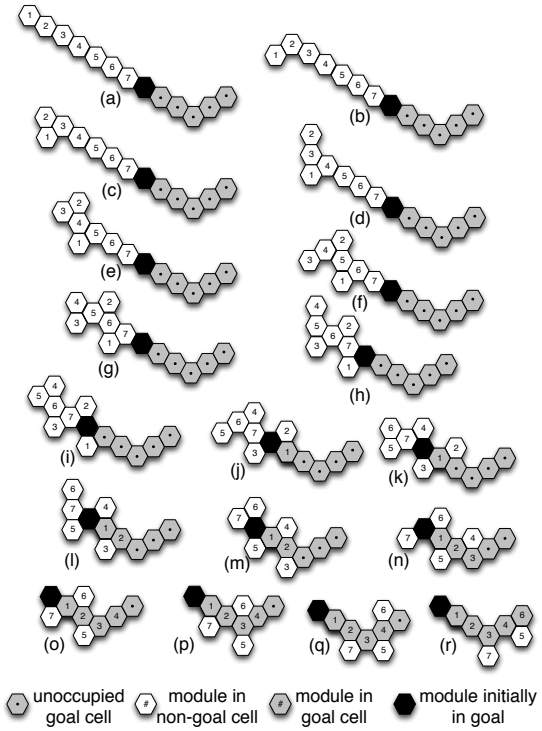


Fig. 5: Illustration of module placement when the path length is odd and *initialSlope* $\neq$ *lastSlope*. Configurations are labeled sequentially (a) through (r). Modules 5 and 7 would fill cells to the south of the SP (not shown).

To see why this algorithm works, observe in Fig. 5 how pairs of moving modules are in the same vertical column at each step. Every time the SP has a bend (which must be either to the NE or SE), the relative ordering of the module position numbers in the SP may change. So, for example, suppose the odd numbered modules start filling the path, each followed by an even numbered module. After the path bends, the even modules may begin filling the path before the odd ones.

Algorithm FILLSUBSTRATEPATH assumes the first *num* (or *num*+1) modules will be the first to enter cells on

the SP. Every obtuse bend followed by a straight chain of two or more modules in the SP changes the sequence of module positions on the path from odd/even to even/odd (and vice versa). If *num* is even, the first *num* modules will enter the path, although not necessarily in position order. If *num* is odd and the last path direction equals the initial path direction, the final module on the path (an odd numbered module) will reach the path before its even numbered counterpart.

The most complicated case is handled by lines 7–8 of FILLSUBSTRATEPATH. In this case, the final bend causes the even numbered module in position *num*+1 to reach the SP before the module in position *num*. In this case, the module in position *num* will fill a goal cell on either the N or S side of the SP (see modules 5 and 7 in Fig. 5, which will fill cells to the S).

5. Distributed Reconfiguration Phase

Each module calculates its rotation direction and delay before moving, after it determines its position in I.

Modules that will end the reconfiguration on the SP do not have to choose final goal destinations because the first module to enter a SP goal cell stops in that cell. However, the algorithm matches the remainder of the modules to goal cells on the N and S of the SP, alternating between CW and CCW rotation. Modules fill the N and S columns from right to left and from the SP northward and southward, like the algorithms presented in [15]. These modules choose their final destination from a list of goal cells that is generated by and is the same at every module.

Single-cell spacing between moving modules is the most efficient movement pattern since a module can't move into a neighboring occupied cell. However, this spacing may cause deadlock problems when the SP forms an acute angle with a goal column (see Fig. 2). To solve this problem, we use a technique like that presented in [7], [8] and [4] to either permanently or temporarily halt particular modules in areas where collision or deadlock may occur.

Each module runs a mapping algorithm in which certain goal cells are marked as WAIT-INDICATOR (WI) or STOP-INDICATOR (SI) cells. Each WI cell has an adjacent empty cell marked as a WAITCELL (WC) and each SI cell has an adjacent goal cell marked as a STOPCELL(SC). Modules mapped to SC cells choose the SC cell as their final destination. A module mapped to a WC cell temporarily halts in the WC cell when the adjacent WI cell is occupied (this only occurs once per WI per reconfiguration). The modules that temporarily stop in WC cells are matched to the extreme N or S cell in the column directly to their right. These modules temporarily stop until all cells but the extreme N or S cell in the column to the right are filled, when they resume movement.

To prevent trapping modules in WC cells, particular goal cells are marked as DELAYSET (DS). Any module mapped to a DS goal cell will delay 3 steps before it begins moving out of the initial chain after the first round in which it becomes FREE. Note that SI and WI markers are only needed when modules will collide in acute angle corners, which cannot happen if the column immediately to the right is short enough (see, for example, the 4th goal column from left side of goal configuration in Fig. 6).

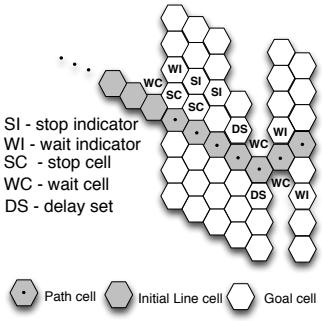


Fig. 6: Placement of markers in each module's map of G to avoid deadlock in acute angle corners.

The modules in WCs or SCs prevent moving modules from making contact in acute angle corners by filling the corner with a module, either permanently (SC) or temporarily (WC). Filling this corner cell allows the rest of the modules in the column immediately to the right to move over the corner with no deadlock.

Below we describe scenarios in which SC and WC cells are used. For the examples given in Figs. 7 and 8, we will concentrate on the columns of G on the north side of the SP (the south side is symmetric).

1) The first time a module i enters an SC in column A when there is an *occupied* adjacent SI cell in column B (as shown in Fig. 7), module i permanently stops in the SC. Column A must be non-empty in this case.
2) The first time a module i enters a WC in column A when there is an *occupied* adjacent WI cell in column B (as shown in Fig. 8), module i delays further movement until all but one goal cell in column A (the N-most cell in this case) is filled. The module in the WC then starts to move again to fill the N-most cell in column B. Column A is empty in this case. When there is a non-empty goal column to the left of empty column A with an associated WC cell, the module matched to the DS goal cell waits for 3 steps after it becomes FREE in the initial line before starting to move. Note that a goal cell may be marked as both a WI and DS.

After the proper cells are marked, each module creates a list of the goal cells, in the order they are to be filled.

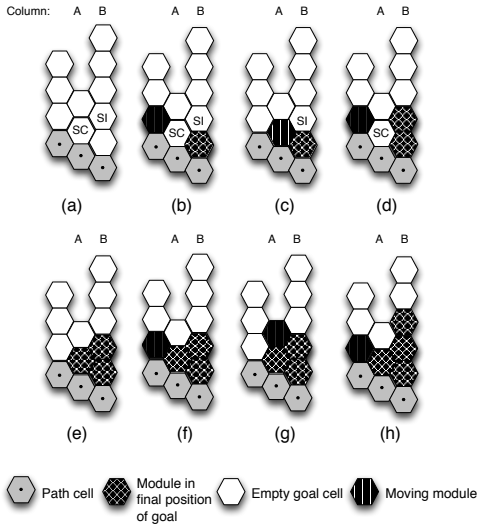


Fig. 7: Segment of N side and SP with cells marked as SI and SC. Figure (a) shows a segment of the map at each module after cells are marked initially; figures (b)–(h) show the consecutive states after modules rotating CW begin filling column B from the left in a S to N fashion. At this point, all N columns to the right of column B are filled. Once the SI cell in column B is occupied, the next module to enter the SC cell in column A stops and column B continues to be filled.

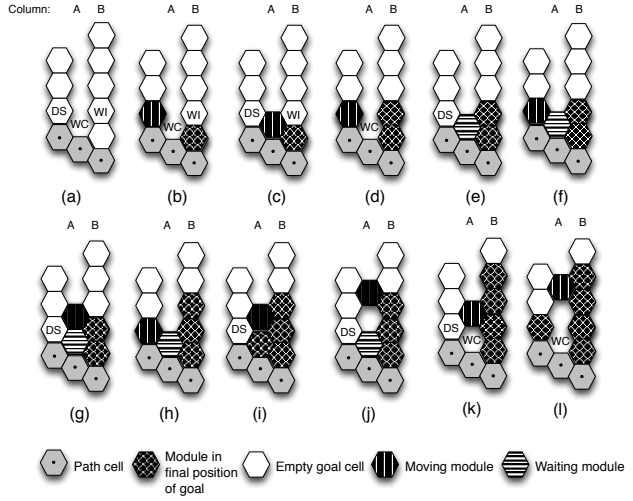


Fig. 8: Segment of N side and SP with cells marked as WI, WC, and DS. Fig. (a) shows the initial map at each module; figures (b)–(l) show the consecutive states after modules rotating CW begin filling column B from the left in a S to N fashion. At the time step shown in part (b), all N columns to the right of column B are filled. Once the WI cell in column B is occupied, the next module to enter the WC cell in column A waits until column B is filled except for the N-most cell. Then the module in the WC begins moving to the N-most cell in column B. Parts (k)–(l) show the delay between the module ending in the DS cell and the module in the WC.

Since modules know their position in I before the start of motion planning, once each module has created the list, each chooses the goal destination that corresponds to their position number. Final destinations and take-off delays are chosen prior to the distributed reconfiguration phase and the timing of temporary module halt periods are determined by each module according to its local environment during distributed reconfiguration.

6. Simulation

In this section, we briefly describe the results of running our new and old algorithms on the same configurations, both for filling the SP and for filling the rest of the goal configuration, using a discrete event simulator. The main difference between the algorithms presented in this paper and the original algorithms is the amount of inter-module spacing between moving modules. In order to avoid deadlock in acute angle corners, the original algorithm required there to be at least 2 empty spaces between moving modules. The new algorithm requires only a single space between moving modules.

We are still in the process of testing the performance of the new versus the old algorithms, but in every trial the number of rounds for the reconfiguration was lower for the new algorithms. Table 1 shows the number of rounds needed for reconfiguration on square goal configurations of increasing size (for example, see Fig. 9). For each configuration of the same size, identical SPs were used.

Table 1: Comparison of algorithms on square configurations.

Number of Modules	Number of Rounds Old	Number of Rounds New
25 (5X5)	71	53
100 (10X10)	282	222
400 (20X20)	1070	833
625 (25X25)	1616	1268

Table 2: Comparison of algorithms on comb configurations.

Number of Modules	Number of Rounds Old	Number of Rounds New
29	115	93
59	237	179
119	477	359

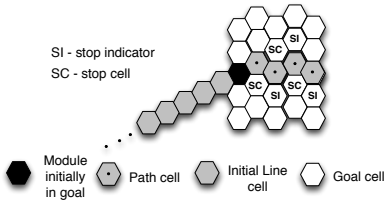


Fig. 9: Square configuration with 25 modules.

When no modules experience a temporary delay, as in the square configurations listed in Table 1, the new algorithms are more efficient than the older ones. The number of rounds used by the new algorithms remained consistent at just over twice the number of modules in the square configurations as the number of modules increased, making the running time linear in the number of modules. The number of rounds used by the old algorithms grows faster than those used by the new algorithms as the number of modules increases.

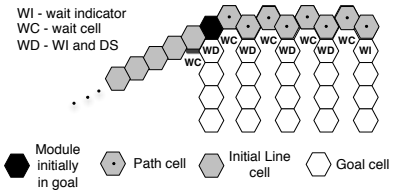


Fig. 10: Comb configuration with 29 modules. Larger combs had same length vertical columns but a longer horizontal backbone.

When some modules experience a temporary delay and the SP does not bisect the goal, as in the comb configuration shown in Fig. 10, the reconfiguration time of the new algorithms is more than double the number of modules, but is still better than the performance of the old algorithms as the number of modules increases (see Table 2).

Even though the new algorithms we present in this paper are more efficient than our previous algorithms, they do not work for as many goal configurations as do the older algorithms. Because they use the conservative two cell inter-module spacing, the older algorithms require fewer constraints on the SP. In particular, the older algorithms work as long as the SP is right-monotone. Moreover, they result in a collision and deadlock free reconfiguration even when the SP contains vertical sections, as long as the vertical sections are separated by at least 3 columns [16]. We do not feel that this difference is a major problem considering that the goal configuration can be separated into segments that end at the first vertical column in the substrate path and the new algorithms can be applied to each such segment, from right to left.

7. Conclusions and future work

We have presented algorithms for filling an arbitrary goal configuration in a column-wise fashion. These algorithms guarantee that collision and deadlock do not occur when run on an admissible goal configuration. The algorithms were shown via simulation to be more efficient (although not asymptotically so) than our older algorithms.

For future work, we are continuing to write algorithms to comprise a complete, deterministic planner for the reconfiguration of hexagonal, metamorphic robots. As part of this complete planner, we are concentrating on deterministic algorithms to reconfigure an arbitrary but admissible shape initial configuration into a straight chain.

References

[1] Z. Butler, K. Kotay, D. Rus, and K. Tomita. Cellular automata for decentralized control of self-reconfigurable robots. In *Proc. of the ICRA 2001 Workshop on Modular Robots*, pages 21–26, 2001.

[2] Z. Butler and D. Rus. Distributed motion planning for modular robots with unit-compressible modules. *Intl. Journal on Robotics Research*, 2003.

[3] G. Chirikjian. Kinematics of a metamorphic robotic system. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 449–455, 1994.

[4] P. Ivanov and J. Walter. Layering algorithm for collision-free traversal using hexagonal self-reconfigurable metamorphic robots. In *Proc. of the IEEE International Conference on Intelligent Robots and Systems*, pages 521–528, 2010.

[5] K. Kotay, D. Rus, M. Vona, and C. McGray. The self-reconfiguring robotic molecule: design and control algorithms. In *Proc. of Workshop on Algorithmic Foundations of Robotics*, pages 376–386, 1998.

[6] T. Larkworthy and S. Ramamoorthy. An efficient algorithm for self-reconfiguration planning in a modular robot. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 5139–5146, 2010.

[7] D. Little and J. Walter. Using hexagonal metamorphic robots to form temporary bridges. In *Proc. of the IEEE International Conference on Intelligent Robots and Systems*, pages 2652–2657, 2005.

[8] S. Matysik and J. Walter. Using a pocket-filling strategy for distributed reconfiguration of a system of hexagonal metamorphic robots in an obstacle-cluttered environment. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 4265–4272, 2009.

[9] Y. Miao, G. Yan, and Z. Lin. A distributed reconfiguration strategy for target enveloping with hexagonal metamorphic modules. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 4804–4809, 2011.

[10] S. Murata, H. Kurokawa, and S. Kokaji. Self-assembling machine. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 441–448, 1994.

[11] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, and S. Kokaji. A 3-d self-reconfigurable structure. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 432–439, 1998.

[12] A. Nguyen, L. J. Guibas, and M. Yim. Controlled module density helps reconfiguration planning. In *Proc. of 4th International Workshop on Algorithmic Foundations of Robotics*, 2000.

[13] A. Pamecha, I. Ebert-Uphoff, and G. Chirikjian. Useful metrics for modular robot motion planning. *IEEE Transactions on Robotics and Automation*, 13(4):531–545, 1997.

[14] D. Rus and M. Vona. Self-reconfiguration planning with compressible unit modules. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 2513–2520, 1999.

[15] J. Walter, B. Tsai, and N. Amato. Algorithms for fast concurrent reconfiguration of hexagonal metamorphic robots. *IEEE Transactions on Robotics*, 21(4):621–631, 2005.

[16] J. Walter, J. Welch, and N. Amato. Concurrent metamorphosis of hexagonal robot chains into simple connected configurations. *IEEE Transactions on Robotics and Automation*, 18(6):945–956, 2002.

[17] J. Walter, J. Welch, and N. Amato. Distributed reconfiguration of metamorphic robot chains. *Springer-Verlag Journal on Distributed Computing*, 17:171–189, 2004.

[18] M. Yim. A reconfigurable modular robot with many modes of locomotion. In *Proc. of Intl. Conf. on Advanced Mechatronics*, pages 283–288, 1993.

[19] M. Yim, J. Lamping, E. Mao, and J. G. Chase. Rhombic dodecahedron shape for self-assembling robots. In *SPL TechReport P9710777, Xerox PARC*, 1997.

[20] Y. Zhang, M. Yim, J. Lamping, and E. Mao. Distributed control for 3d shape metamorphosis. *Autonomous Robots Journal, special issue on self-reconfigurable robots*, 2000.

Reducing Fragment Oscillation of Dynamic Fragment Allocation in Non-Replicated Distributed Database System

Tarun Gulyani and Pallath Paul Varghese
Pitney Bowes Software
India

Abstract— *Production and consumption of data has seen an exponential increase in recent times thus necessitating a solution which addresses the issues related to physical storage, reliability and accessing speed limits of centralized system. Distributed Database System is the solution, which overcomes these limits [1]. Designing Distributed Database System has many issues, and one of them is fragment allocation/movement. Various fragment allocation algorithms already exist in Distributed Database System. There are strengths and shortcomings of these algorithms. This paper gives the brief overview of these algorithms and proposes one new algorithm (Reduced Fragment Oscillation Algorithm). RFO algorithm moves the fragment from source node to target node by considering both the frequency of fragment access by region as well as individual nodes. It increases the overall system performance. RFO algorithm also reduces the amount of topological data required in decision making.*

1. Introduction

Distributed Database System is the Database System in which data is stored at multiple computers (node) either placed at the same physical location or spread over several geographical locations. These computers are connected to some network, via internet, intranet or extranet. Demands of Distributed databases are increasing rapidly than ever before, because it overcomes the limitations of centralized system [1]. Distributed database hides the distribution of logical and physical components of databases from users.

The prime motivation in distributed database systems are to improve performance, to increase the availability, expandability and access facility of data [8]. In Distributed database the major challenge is how to distribute the data in the best manner. If the architecture of distributed database is not perfect, then its performance will be worse in comparison to centralized database architectures.

Although distributed systems are highly desirable, the heterogeneity and lack of adherence to standards, makes it difficult to build a proper functioning system. Complexity in designing distributed database architecture is in maintaining multiple disparate systems instead of one big centralized system. Different design techniques are used to maintain the consistency in Distributed Database Architectures. Various design techniques used in Distributed Database System are replication, duplication, fragmentation, local autonomy, synchronous and asynchronous techniques etc. In case of replication technique any change in the database is replicated over all the databases stored over multiple nodes. This process is significantly time consuming and requires high network bandwidth. In duplication technique, a copy of the master database is regularly maintained in another database but it is a simple process compared to replication. In local autonomy technique, each node of distributed database can define their own policy. In asynchronous technique, a constraints of time, execution latencies and message latencies does not exist. On the other hand synchronous technique have both upper and lower bound for time constraint, execution latencies and message latencies. Depending upon the requirement for distributed databases, these technologies are used. Major concern in Distributed Database System is how to architect a good design for fragmentation of data and data allocation [2,9]. Data fragmentation as the word suggest is to distribute the data into fragments over multiple nodes either physically co-located or present at several geographical locations.

Data fragmentation technique helps increase the efficiency of data access/query performance, as it enables a mechanism to store the fragment of data at a node where it is frequently used. It increases the parallelism, because large transaction can be divided into sub transaction and all sub transaction start processing concurrently. Data fragmentation can be horizontal fragmentation, vertical fragmentation or mixed fragmentation [7]. Horizontal fragmentation can be achieved using either by range or hash fragmentation [3].

These fragmentation techniques work on static environment, in which data access pattern and/or query pattern is static. For dynamic environment, where access pattern change dynamically, static allocation decreases performance. Data fragmentation algorithms used in dynamic environment include Optimal algorithm [6], Threshold algorithm [11], NNA (Near Neighborhood Allocation) algorithm [4], BGBR algorithm [5], FNA (Fuzzy Neighborhood Allocation) algorithm [10] etc. Main objective of these algorithm is to minimize the access cost as well as data transfer cost in executing the set of queries.

A brief overview of existing algorithms is presented in the following sections and various notations/terms used in the entire paper are listed in TABLE 1.

2. Existing Algorithms

In Optimal algorithm [6], a static environment method is used to initially distribute the fragments to all nodes in the distributed network and an optimal algorithm is executed at each node. This algorithm depends highly on the frequency of access. The transfer of fragments from one node to the other is influenced by the change in frequency of access of fragments by each node. And if the frequency of access of a fragment changes frequently then there is an increase in the cost of transfer of fragment and network traffic. This would also result into significantly high oscillations of fragments between nodes. In this algorithm the access time of the fragment with highest frequency of access is significantly reduced but has a negative effect on the access time for the other nodes. The reason being the transfer of the fragment between nodes does not take the network topology into consideration.

Table 1
NOTATIONS

Node	Node is the physical locations of database, where data of distributed database stored. Each node contains multiple fragments. Number of fragments at each node can vary.
Source Node	Source node is the node, from where accessing fragment transfer to other node.
Target Node	Target node is the node, which has highest access count to fragment of source node.
Neighbor Node	Neighbor node is the node, which is in between the path of source node to target node and also nearest to source node.
Region	Group of Nodes makes Region, where each node stored data of distributed database.
Ni	Ni is known as ith Node.
F	Whole unit of database at each node is known as F, which contains more than one Fragment. F=F1 + F2 + F3...+ Fk. Value of k can vary at each node.
A	Node-Fragment Access counter matrix. Where each element a_{xy} represents, how many times fragment x access by node y.
R	Region-Fragment Access counter matrix. Where each element r_{xz} represents, how many times fragment x access by Region z.
Threshold t	Predetermined Region-Fragment access count value. Whenever Region-Fragment access count reaches to this value, fragment moves from source node.

Storage space requirement of an Optimal algorithm [6] is more as each fragment has to store the access counter value corresponding to each node. The heuristic threshold algorithm [11] addresses this drawback.

In the threshold algorithm a counter value at each fragment is initialized to 0. The counter value is incremented, whenever remote access to the fragment occurs, and its value is reset to 0 when local access to the fragment occurs. Counter value is incremented only at remote access and not at local access. When counter value of fragment exceeds the predefine threshold value, then the ownership of the fragment is transferred to the remote node that recently accessed the fragment. If the threshold value is t, then this algorithm assures that the fragment remains at the new node for at least t+1 accesses [11]. Threshold value plays a major role in this algorithm. If threshold value more, then migration of fragments at the node is less. But if threshold value is less, then migration of fragments at the node is more. Approach of this algorithm is heuristic, as the ownership of the fragment is transferred to the node which has recently accessed the fragment which may or may not be node which has highest access to the fragment. The algorithm does not use the knowledge of topology so optimal node chosen for migration may not be the globally optimal node.

Near Neighborhood Allocation (NNA) [4] algorithm is a variation of Optimal Algorithm where finding the node (say target node) to which the fragment needs to be transferred from the original node (say source node) is same as optimal algorithm. However the fragment is transferred to the nearest neighbor of the source node which is in the path between the source node and target node. Routing algorithm is used to find the nearest neighbor of the source node Here the knowledge of network topology is taken into consideration while selecting the node for transfer of fragment. NNA algorithm avoids fragment oscillation, which occur in optimal algorithm and also avoids multihop transfers. This also helps in reducing the access time from the target node. The NNA performs better than Optimal Algorithm when the size of fragment and size of network is very large. However in NNA algorithm, by moving fragment from source node to neighboring node, decreases the delay in access time for only those nodes which are in the path of between source and target nodes but for others node this may increase. If fragment access count value of the nodes, which are not in the path between target and source nodes is much more than nodes in the path then performance of system decreases. NNA algorithm avoids oscillation, so it reduces the consumption of network resources. For small fragment size, optimal algorithm is better than NNA algorithm, because cost of movement of small fragment is not more than access cost.

An improved version of NNA algorithm called FNA (Fuzzy Neighborhood Allocation) algorithm [10] is able to detect oscillation conditions and provides a solution to prevent redundant fragment migration. FNA uses a fuzzy inference engine to detect oscillations in fragment requests and ignore fragment migrations.

BGBR Algorithm [5] is an improvement on NNA algorithm. It reduces the response time and fragment migration time from source node to target node compared to NNA algorithm. BGBR algorithm requires the knowledge of the complete network topology for migrating fragment from source node to target node. It avoids fragment oscillation. Although NNA algorithm reduces oscillation by knowledge of topology but it does not use the complete knowledge of topology. In NNA algorithm for transfer of fragment from source node to neighboring node, the shortest path does not give the

assurance of best path according to global topology. BGBR performs better in both small as well as large fragment sizes however NNA performs better only when size of fragment is large. BGBR reduces fragment mobility as compared to Optimal, Threshold and NNA algorithm. BGBR algorithm shows better result, but it requires significant amount of calculations and also it requires capturing of significant amount of data about network topology.

Ulus et. al. [12] has proposed an algorithm based on Markov Chain Model where the each node autonomously decides whether to transfer the ownership of a fragment in DDS to another node or not and it depends on the past accesses of the fragment. Each fragment continuously migrates from the node where it is not accessed locally more than a certain number of past accesses, namely a threshold value.

A new dynamic fragment allocation algorithm in Non-Replicated allocation scenario proposed by Nilarun [8], incorporates access threshold time constraint of database access and most importantly the volume of data transmitted to dynamically reallocate fragments to sites at runtime in accordance with the changing access probability of nodes to fragments .

The next section discusses the proposed algorithm which addresses few drawbacks in the existing approaches of Dynamic Fragment Allocation in Non-Replicated Distributed Database System.

3. Reduced Fragment Oscillation (RFO) Algorithm

The proposed algorithm (RFO Algorithm) for dynamic allocation of data in non replicated distributed database system is designed to address the issues in the existing approaches where the movement of the fragment depends only on the node which has the highest frequency of access to a fragment. But in RFO algorithm consideration is also given to the highest fragment accessing region instead of considering the target there by reducing the access time from more than one node.

Steps of Reduced Fragment Oscillation (RFO) Algorithm are:

Step1: Group all the nodes into regions, with each region having approximately equal number of nodes subject to the constraint that the nodes are in near proximity to each other.

Figure 1 shows the layout of the regions and the access patterns of each node in the region. The steps of grouping the nodes into different regions is given below:

Step 1_a: Decide the value of m, through which whole network will divide in regions.

Step 1_b: Calculate the physical location of database in degree term by projection on map.

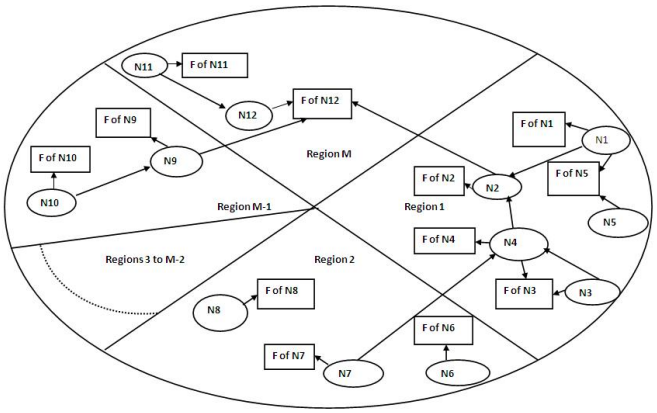


Fig. 1

DIVIDE THE WHOLE REGION OF FRAGMENT ACCESSING NODES

Step 1_c: According to value of m (Number of region), The region to which database belongs is given by: ((Degree of physical location of database / (360^0 / m)) + 1).

Number of fragments at each node can vary. Each fragment can be accessed by local node and/or global (remote) nodes. Global nodes can access the fragment at a node either directly or by more than one hop.

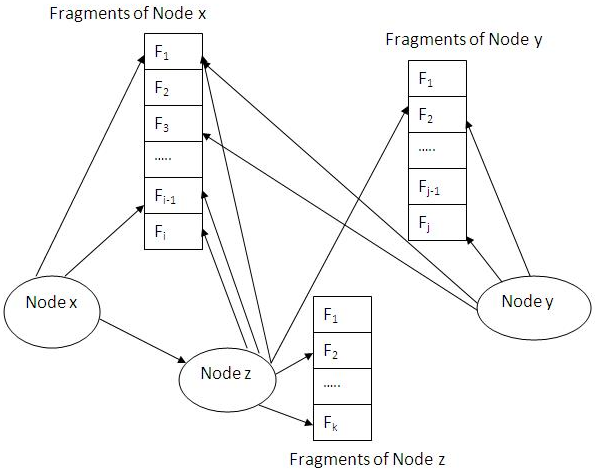


Fig. 2

FRAGMENTS INTERACTION WITH NODES.

Figure 2 shows Node x contains fragments F_1 to F_i, Node y contains F_1 to F_j and Node z contains F_1 to F_k. Number of fragments at each node can vary. Each fragment can be accessed by the local node and/or remote nodes. For example fragment F_1 of Node x is accessed locally by Node x as well as remotely accessed by Node y and Node z. A fragment can be accessed directly or indirectly by more than

one hops. Here fragment F_2 of Node y is accessed by Node x through Node z.

Step2: For each locally stored fragment initialize the Node-Fragment Access Counter matrix A as a_{xy}=0, where x is index of all fragments and y is index of all nodes.

Step3: For each locally stored fragment initialize the Region-Fragment Access Counter matrix R as r_{xz}=0, where x is index of all fragments and z is index of all regions.

Step4: Check access request for each stored fragment from both local node and remote nodes.

Step5: Fragment Access Counter matrix A, increases the access counter value for access of fragments from a node. If node y access the fragment x then a_{xy}=a_{xy}+1.

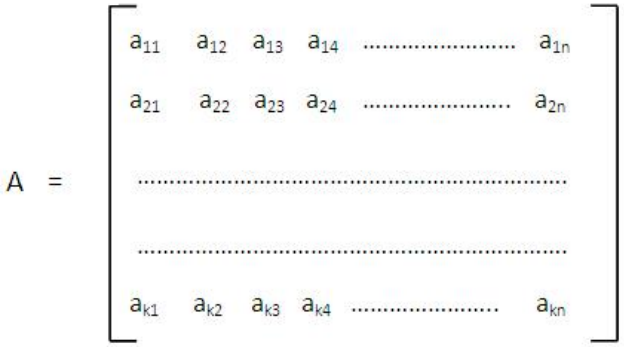


Fig. 3
NODE-FRAGMENT ACCESS COUNTER MATRIX A.

Figure 3 shows Node-Fragment Access Counter matrix A is the size of k by n, where k denote the number of fragments and n denotes the number of nodes.

Step6: Region-Fragment Access counter matrix R, increases the access counter value for accessing fragments from the regions. If region z access the fragment x then r_{xz}=r_{xz}+1.

Figure 4 shows Region-Fragment Access counter matrix R is the size of k by m, where k denotes the number of fragments and m denotes the number of regions.

Figure 5 shows Node contains local database F, which consist of more than one fragment. It contains Node-Fragments Access counter matrix A and Region-Fragment Access counter matrix R.

Step7: Migrate the fragment from source to the target in the region which has max count in R and then in that region

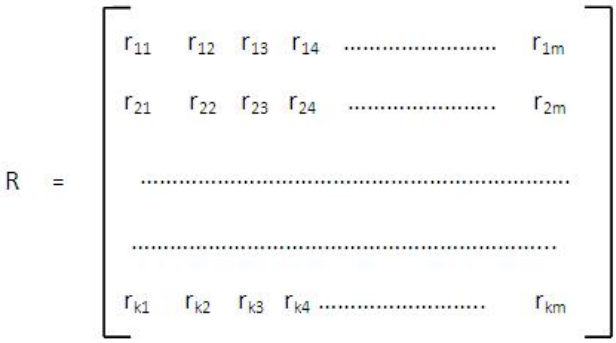


Fig. 4
REGION-FRAGMENT ACCESS COUNTER MATRIX R.

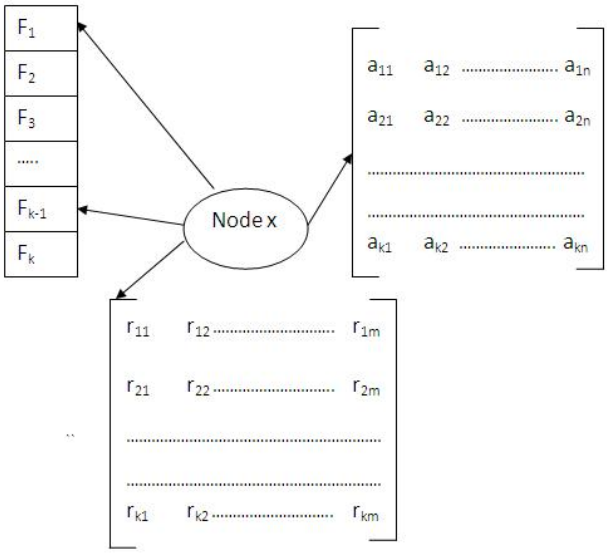


Fig. 5
INFORMATION AT EACH NODE.

move the fragment to the node in that region which has max count in A.
The algorithm only considers fragments at each node which has Region-Fragment Access counter value greater than a predefined threshold t.

Figure 6, 7 and 8 explains the process of identifying the target node for migrating the fragment.

Figure 6 shows F_k is one of the fragment of Node N13 in Region 3, which is accessed by Nodes (N1,N2,N3,N4 and N5) from Region 1,(N6,N7,N8) from Region 2, (N9,N10) from Region M-1 and (N11,N12) from Region M.

Figure-7 shows only rows of Node-Fragment and Region-Fragment Access Counter matrix of Fragment F_k according to access in Figure-6 .

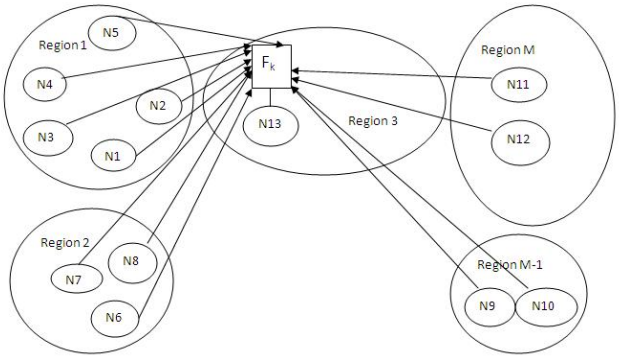


Fig. 6
FRAGMENT F_k ACCESS FROM ALL REGIONS.

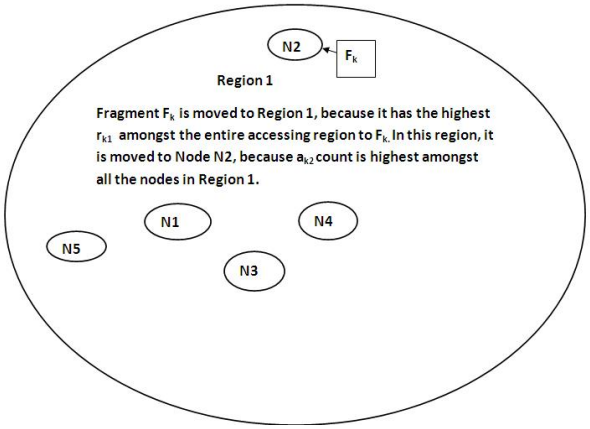


Fig. 8
FRAGMENT MOVED TO HIGHEST ACCESSING REGION.

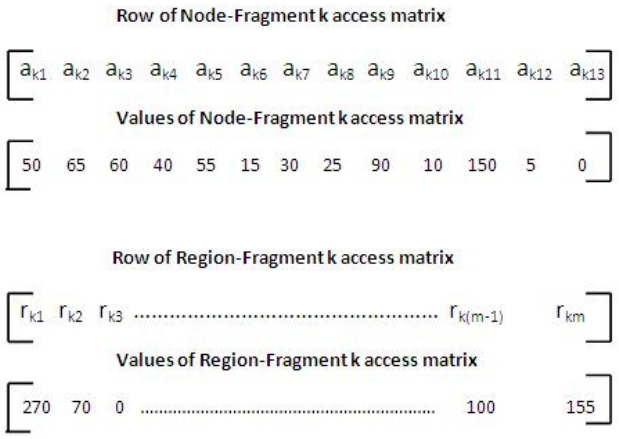


Fig. 7
ROWS OF NODE-FRAGMENT AND REGION-FRAGMENT ACCESS COUNTER MATRIX OF FRAGMENT F_k.

The Region-Fragment access count for each region is calculated. The region which has the maximum count is chosen as the region where the fragment would be moved.
Region 1 has the highest access count to Fragment F_k in comparison to other regions. Therefore Fragment F_k moved to Region 1. In Region 1 the Node-fragment access count for all the nodes is computed for the Fragment F_k.The node which has the maximum count is chosen as the target node. Therefore (in the current example Figure 8) within Region 1, Fragment F_k moved to Node N2.
If Optimal/Threshold algorithm is used instead of RFO algorithm then fragment F_k will be moved to Node N11 in Region M. Because it has the maximum Node-Fragment access count of 150, which is greater than that of all other nodes.
Transferring the Fragment F_k to Node N11 in Region M decreases the access time of node N11 for the fragment F_k, but increase the access time to all the nodes(N1,N2,N3,N4,N5) of Region 1. Resulting in decrease in total access time for the fragment F_k from Region M, which had Region-Fragment access count of 155, but increasing the access time of the fragment F_k from Region 1, which had Region-Fragment access count of 270.
RFO algorithm addresses this issue and therefore it reduces the global access time of fragment F_k by transferring the fragment F_k to node N2 instead of node N11.

4. Conclusion

Proposed algorithm decreases the migration/oscillations of fragments in comparison to optimal, and threshold algorithm using the knowledge of network topology which is not the case with both Optimal and Threshold. In comparison to NNA algorithm, RFO algorithm moves the fragment from source node to destination target node by considering both the frequency of fragment access by region as well as individual nodes, there by increasing the overall system performance. RFO algorithm also reduces the amount of topological data required in decision making in comparison to BGBR algorithm.
RFO algorithm depends on two parameters. One is the choice of threshold value which also directly impact fragment oscillation. Another parameter is division of whole network into m regions, where each region contains approximately equal nodes.
RFO Algorithm is suitable for Distributed database architectures where the access pattern to a fragment changes dynamically and distributed database is not replicated. This algorithm significantly minimizes the data transfer and also decreases the amount of complexity required in identifying a suitable target node.

References

[1] ADIBA, M., CHUPIN, J. C., DEMOLOMBE, R., GARDARIN, G., AND BIHAN, J. L. Issues indistributed data base management systems: A technical overview. In Proceedings of the 4th Internationul Conference on Very Large Data Bases (West Berlin, Sept. 1978), pp. 89-110.

[2] Arjan Singh and K.S. Kahlon "Non-replicated Dynamic Data Allocation in Distributed Database Systems" in Proc. IJCSNS International Journal of Computer Science and Network Security, pp. 176 VOL.9 No.9, September 2009.

[3] B. Yang, S. Agrawal and V. Narasayya "Integrating Vertical and Horizontal Partitioning into Automated. Physical Database Design," Proc. 2004 ACM SIGMOD International Conf. Management of Data, pp. 359-370, 2004.

[4] Basseda, R., Tasharofi, S., Rahgozar, M., Near Neighborhood Allocation (NNA): A Novel Dynamic Data Allocation Algorithm in DDB, In proceedings of 11th Computer Society of Iran Computer Conference (CSICC2006), Tehran, 2006.

[5] Bayati, A.; Ghodsnia, P.; Basseda, R.; Rahgozar, M., "A Novel Way of Determining the Optimal Location of a Fragment in a DDBS". Proc. Systems and Networks Communications, 2006. ICSNC '06, pp. 64-64.

[6] John, L. C., A Generic Algorithm for Fragment Allocation in Distributed Database Systems, ACM, 1994.

[7] Navathe, S. B., Ceri, S., Wiederhold, G., and Dou, J., Vertical Partitioning Algorithms for Database Design ACM Transaction on Database Systems, 9, 1984, 680-710.

[8] Nilarun Mukherjee, "Non-Replicated Dynamic Fragment Allocation in Distributed Database Systems ", Communications in Computer and Information Science, 2011, Volume 131, Part 4, 560-569, DOI: 10.1007/978-3-642-17857-3-55.

[9] P. M. G, APERS. Data allocation and distributed query processing. ACM Transactions on Database Systems, vol. 13, No. 3, September 1988, 263-304.

[10] Reza Basseda, Maseud Rahgozar and Caro Lucas, "Fuzzy Neighborhood Allocation (FNA): A Fuzzy Approach to Improve Near Neighborhood Allocation in DDB". Proc. Communications in Computer and Information Science, 2009, Volume 6, Part 2, 834-837, DOI: 10.1007/978-3-540-89985-3-114.

[11] T. Ulus and M. Uysal, "Heuristic Approach to Dynamic Data Allocation in Distributed Database Systems", Pakistan Journal of Information and Technology, 2(3): pp. 231-239, 2003.

[12] T. Ulus and M. Uysal, "A Threshold Based Dynamic Data Allocation Algorithm- A Markove Chain Model Approach", Journal of Applied Science 7(2), pp. 165-174, 2007.

Parallel simulated annealing for the covering arrays construction problem

Himer Avila-George[1], **Jose Torres-Jimenez**[2] **and Vicente Hernández**[1]
[1]Instituto de Instrumentación para Imagen Molecular (I3M). Centro mixto CSIC - Universitat Politècnica de València - CIEMAT, camino de Vera s/n, 46022 Valencia, Spain.
[2]Information Technology Laboratory, CINVESTAV-TAMAULIPAS, Km. 5.5 Carretera Victoria-Soto La Marina, 87130 Victoria Tamps., Mexico.

Abstract—*A covering array (CA) is a combinatorial structure specified as a matrix of N rows and k columns over an alphabet on v symbols such that for each set of t columns every t-tuple of symbols is covered at least once. Given the values of t, k, and v, the optimal covering array construction problem (CAC) consists in constructing a $CA(N;t,k,v)$ with the minimum possible value of N. There are several reported methods to attend the CAC problem, among them are: direct methods, recursive methods, greedy methods, and metaheuristics methods. In this paper, three parallel approaches for simulated annealing, i.e. the independent, semi-independent and cooperative searches are applied to the CAC problem. The empirical evidence supported by statistical analysis indicate that the cooperative approach offers the best execution times and the same upper bounds than the independent and semi-independent approaches. Extensive experimentation was carried out, using 96 well-known benchmark instances, for assessing its performance with respect to the best-known bounds reported previously. The results show that cooperative approach attains 78 new bounds and equals the solutions for other 6 instances.*

Keywords: Covering Array, Simulated Annealing, Parallel Computing

1. Introduction

A covering array, denoted by $CA(N;t,k,v)$ is a matrix $\mathcal{M}$ of size $N \times k$ which takes values from the set of symbols $\{0,1,2,\ldots,v-1\}$ (called the alphabet), and every submatrix of size $N \times t$ contains each tuple of symbols of size t (or t-tuple), at least once. The value N is the number of rows of $\mathcal{M}$, k is the number of parameters, where each parameter can take v values and the interaction degree between parameters is described by the strength t. Each combination of t columns must cover all the v^t t-tuples. Given that there are $\binom{k}{t}$ sets of t columns in $\mathcal{M}$, the total number of t-tuples in $\mathcal{M}$ must be $v^t\binom{k}{t}$. When a t-tuple is missing in a specific set of t columns we will refer to it as a missing t-wise combination. Then, $\mathcal{M}$ is a covering array if the number of missing t-wise combinations is zero.

When a matrix has the minimum possible value of N to be a $CA(N;t,k,v)$, the value N is known as the Covering Array Number. The Covering Array Number is formally defined as (1):

$$CAN(t,k,v) = \min\{N : \exists\ CA(N;t,k,v)\}. \quad (1)$$

Given the values of t, k, and v, the optimal covering array construction problem (CAC) consists in constructing a $CA(N;t,k,v)$ such that the value of N is minimized.

A major application of covering arrays (CAs) arises in software interaction testing, where a covering array can be used to represent an interaction test suite as follows. In a software test we have k components or factors. Each of these has v values or levels. A test suite is an $N \times k$ array where each row is a test case. Each column represents a component and a value in the column is the particular configuration. By mapping a software test problem to a covering array of strength t we can guarantee that we have tested, at least once, all t-way combinations of component values. Thus, software testing costs can be substantially reduced by minimizing the number of test cases N in the covering array. Please observe that software interaction testing is a black-box testing technique, thus it exhibits weaknesses that should be addressed by employing white-box testing techniques. For a detailed example of the use of covering arrays in software interaction testing the reader is referred to [1].

Because of the importance of the construction of (near) optimal covering arrays, much research has been carried out in developing effective methods for construct them. There are several reported methods for constructing these combinatorial models. Among them are: direct methods [2], [3], recursive methods [4], [5], [6], greedy methods [7], [8], [9], [10], [11], and metaheuristics methods [12], [13], [14], [15], [16], [17].

In this paper, we aim to develop an improved implementation of a simulated annealing algorithm for constructing covering arrays. Simulated annealing algorithm is a general-purpose stochastic optimization technique that has proved to be an effective tool for approximating globally optimal solutions to many optimization problems. However, one of the major drawbacks of the technique is its very slow convergence. To address this drawback, we propose three parallel simulated annealing approaches to solve the

CAC problem. The objective is to find the best bounds to some ternary covering arrays by using parallelism. To our knowledge the application of parallel simulated annealing to the CAC problem has not been reported in the literature. A parallel algorithm for the verification of covering arrays is presented in [18]. The methods of parallelization of simulated annealing are discussed in [19], [20], [21], [22], [23].

Contrary to existing simulated annealing implementations [12], [13], our algorithm has the merit of improving two key features that have a great impact on its performance: an efficient heuristic to generate good quality initial solutions and a compound neighborhood function which combines two carefully designed neighborhood relations.

The remainder of this paper is organized as follows: Section 2 describes the components of our sequential annealing algorithm, in Section 3 three parallel simulated annealing approaches are discussed, Section 4 describes the experimental results, and finally, Section 5 presents the conclusions derived from the research presented in this paper.

2. Sequential simulated annealing

In this section, we briefly review simulated annealing (SA) algorithm and propose an implementation to solve the CAC problem.

SA is a randomized local search method based on the simulation of annealing of metal. The acceptance probability of a trial solution is given by (2), where T is the *temperature* of the system, $\Delta\mathcal{C}$ is the difference of the costs between the trial and the current solutions (the cost change due to the perturbation), (2) means that the trial solution is accepted by nonzero probability $e^{-\Delta\mathcal{C}/T}$ even though the solution deteriorates (*uphill move*). In this case, because the cost of the solution increases, we call this update on the solution an *uphill move*.

$$(P) = \begin{cases} 1 & if\ \Delta\mathcal{C} < 0 \\ e^{-\Delta\mathcal{C}/T} & otherwise \end{cases} \quad (2)$$

Uphill moves enable the system to escape from the local minima; without them, the system would be trapped into a local minimum. Too high of a probability for the occurrence of uphill moves, however, prevents the system from converging. In SA, the probability is controlled by temperature in such a manner that at the beginning of the procedure the temperature is sufficiently high, in which a high probability is available, and as the calculation proceeds the temperature is gradually decreased, lowering the probability [24].

In order to use the simulated annealing metaheuristic for a combinatorial optimization problem in particular, there are a number of decisions to be taken. Johnson et al. [25] classified these decisions as follows:

- *Problem-specific choices*:
  - The problem must be clearly formulated, so that the set of feasible solutions is defined.
  - The neighborhood of any solution must also be defined as well as a way of determining the value of the objective to be minimized.
  - An initial solution must also be generated.
- *Generic choices*:
  - *Initial Temperature*
  - *Cooling schedule*, a temperature function, $T(t)$, to determine how the temperature is to be changed
  - *Maximum neighboring solutions per temperature*, viz. the number of iterations, $N(t)$, to be performed at each temperature
  - *A stopping criterion to terminate the algorithm*

The following paragraphs will describe each of the components of the implementation of our SA. The description is done given the matrix representation of a covering array.

2.1 Internal representation

A covering array can be represented as a matrix M of size $N \times k$, where the columns are the parameters and the rows are the cases of the test set that is constructed. Each cell $m_{i,j}$ in the array accepts values from the set $\{1, 2, \ldots, v_j\}$ where v_j is the cardinality of the alphabet of j-th column. The size of the search space M is then given by (3):

$$|M| = v^{Nk}. \quad (3)$$

2.2 Initial solution

The *initial solution* M is constructed by generating M as a matrix with maximum Hamming distance. The Hamming distance $d(x, y)$ between two rows $x, y \in M$ is the number of elements in which they differ. Let r_i be a row of the matrix M. To generate a random matrix M of maximum Hamming distance the following steps are performed:

1) Generate the first row r_1 at random.
2) Generate s rows $c_1, c_2, \ldots, c_s$ at random, which will be candidate rows.
3) Select the candidate row c_i that maximizes the Hamming distance according to

$$g(r_i) = \sum_{s=1}^{i-1} \sum_{v=1}^{k} d(m_{s,v}, m_{i,v}),$$

$$\textbf{where } d(m_{s,v}, m_{i,v}) = \begin{cases} 1 \textbf{ if } m_{s,v} \neq m_{i,v} \\ 0 \textbf{ Otherwise} \end{cases}$$

and added to the i-th row of the matrix M.
4) Repeat from step 2 until M is completed.

An example is shown in Fig. 1. The number of symbols different between rows r_1 and c_1 are 4 and between r_2 and c_1 are 3 summing up 7. Then, the hamming distance for the candidate row c_1 is 7.

$$Rows \begin{cases} r_1 = \{\ 2 \quad 1 \quad 0 \quad 1\ \} \\ r_2 = \{\ 1 \quad 2 \quad 0 \quad 1\ \} \\ c_1 = \{\ 0 \quad 2 \quad 1 \quad 0\ \} \end{cases}$$

$$Distances \begin{cases} d(r_1, c_1) = 4 \\ d(r_2, c_1) = 3 \\ g(c_1) = 7 \end{cases}$$

Fig. 1: Example of the hamming distance between two rows r_1, r_2 that are already in the matrix $\mathcal{M}$ and a candidate row c_1.

2.3 Evaluation function

The *evaluation function* $C(M)$ is used to estimate the goodness of a candidate solution. Previously reported metaheuristic algorithms for constructing covering arrays have commonly evaluated the quality of a potential solution (covering array) as the number of combination of symbols missing in the matrix M [13], [14], [15]. Then, the expected solution will be zero missing. In the proposed SA implementation this evaluation function definition was used. Its computational complexity is equivalent to (4):

$$O\left(N\binom{k}{t}\right). \tag{4}$$

2.4 Neighborhood function

Given that SA is based on Local Search (LS) then a neighborhood function must be defined. The main objective of the neighborhood function is to identify the set of potential solutions which can be reached from the current solution in a local search (LS) algorithm. In case two or more neighborhoods present complementary characteristics, it is then possible and interesting to create more powerful compound neighborhoods. The advantage of such an approach is well documented in [26]. Following this idea, and based on the results of our preliminary experimentations, a neighborhood structure composed by two different functions is proposed for this SA algorithm implementation.

Two *neighborhood functions* were implemented to guide the local search of our SA algorithm. The neighborhood function $\mathcal{N}_1(s)$ makes a random search of a missing t-tuple, then tries by setting the j-th combination of symbols in every row of M. The neighborhood function $\mathcal{N}_2(s)$ randomly chooses a position (i, j) of the matrix M and makes all possible changes of symbol. During the search process a combination of both $\mathcal{N}_1(s)$ and $\mathcal{N}_2(s)$ neighborhood functions is employed by our SA algorithm. The former is applied with probability ρ, while the latter is employed at a $(1-\rho)$ rate. This combined neighborhood function $\mathcal{N}_3(s, x)$ is defined in (5), where x is a random number in the interval [0, 1).

$$\mathcal{N}_3(s, x) = \begin{cases} \mathcal{N}_1(s) & \text{if } x \leq \rho \\ \mathcal{N}_2(s) & \text{if } x > \rho \end{cases} \tag{5}$$

2.5 Cooling schedule

The *cooling schedule* determines the degree of uphill movement permitted during the search and is thus critical to the SA algorithm's performance. The parameters that define a cooling schedule are: an initial temperature, a final temperature or a stopping criterion, the maximum number of neighboring solutions that can be generated at each temperature, and a rule for decrementing the temperature. The literature offers a number of different cooling schedules, see for instance [19], [27]. In our SA implementation we preferred a geometrical cooling scheme mainly for its simplicity. It starts at an initial temperature T_i which is decremented at each round by a factor α using the relation shows in (6). For each temperature, the maximum number of visited neighboring solutions is L. It depends directly on the parameters (N, k, and v) of the studied covering array. This is because more moves are required for covering arrays with alphabets of greater cardinality.

$$T_k = \alpha T_{k-1} \tag{6}$$

2.6 Termination condition

The *stop criterion* for our SA is either when the current temperature reaches T_f, when it ceases to make progress, or when a valid covering array is found. In the proposed implementation a lack of progress exists if after ϕ (frozen factor) consecutive temperature decrements the best-so-far solution is not improved.

2.7 Simulated annealing pseudocode

The Algorithm 1 presents the SA heuristic as described above. The meaning of the three functions is obvious: INITIALIZE computes a start solution and initial values of the parameters T and L; GENERATE selects a solution from the neighborhood of the current solution, using the neighborhood function $\mathcal{N}_3(s, x)$; CALCULATE_CONTROL computes a new value for the parameter T (cooling schedule) and the number of consecutive temperature decrements with no improvement in the solution.

Unlike the classical method which takes as a solution to the problem, the last value obtained in the annealing chain. We memorize the best solution found during the whole annealing process (see lines 3 and 12).

In the next section, it is presented three parallel SA approaches for solving the CAC problem.

Algorithm 1 Sequential SA approach for the CAC problem.

```
1: function SA( )
2:     INITIALIZE(M, T, L)
3:     M* ← M
4:     repeat
5:         for i ← 1 to L do
6:             M_i ← GENERATE(M)
7:             ΔC ← C(M_i)-C(M)
8:             x ← random                ▷ x in the range [0,1)
9:             if ΔC < 0 or e^{-ΔC/T} > x then
10:                M ← M_i
11:                if C(M) < C(M*) then
12:                    M* ← M
13:                end if
14:            end if
15:        end for
16:        CALCULATE_CONTROL(T, φ)
17:    until stop_criterion
18: end function
```

3. Parallel simulated annealing

Parallelization is recognized like a powerful strategy to increase algorithms efficiency; however, SA parallelization is a hard task because it is essentially a sequential process.

In evaluating performance of a parallel simulated annealing (PSA), it needs to consider solution quality as well as execution speed. The execution speed may be quantified in terms of *speed-up* ($\mathcal{S}$) and *efficiency* ($\mathcal{E}$). The $\mathcal{S}$ is defined as the ratio of the execution time (on one processor) by the sequential SA to that by the PSA (on P processors) for an equivalent solution quality. In the ideal case, $\mathcal{S}$ would be equal to P. The $\mathcal{E}$ is defined as the ratio of the actual $\mathcal{S}$ to the ideal $\mathcal{S}(P)$.

Next, we propose three parallel implementations of the SA algorithm described in Section 2. For these cases, let P denote the number of processors and L the length of Markov chain.

3.1 Independent search approach

A common technique to parallelizing SA is the *independent search approach* (IS) [19], [21], [28]. In this approach each processor independently perturbs the configuration, evaluates the cost, and decides on the perturbation. The processors P_i, $i = 0, 1, \ldots, P-1$, carry out the independent annealing searches using the same initial solution and cooling schedule as in the sequential algorithm. At each temperature P_i executes $N \times k \times v^2$ annealing steps. When each processor finishes, it sends its results to processor P_0. Finally, processor P_0 chooses the final solution among the local solutions.

3.2 Semi-independent search approach

Our *semi-independent search approach* (SS) is an implementation of the *division algorithm* [19]. In the division algorithm, parallelism is obtained by dividing the effort of generating a Markov chain over the available processors. A Markov chain is divided into P sub-chains of the length $\lfloor L/P \rfloor$. In this approach, the processors exchange local information including intermediate solutions and their costs. Then, each processor restarts from the best intermediate ones.

Compared to the IS, communication overhead in this SS approach would be increased. However, each processor can utilize the information from other processors such that the decrease in computations and idle times can be greater than the increase in communication overhead. For instance, a certain processor which is trapped in an inferior solution can recognize its state by comparing it with others and may accelerate the annealing procedure. That is, processors may collectively converge to a better solution.

3.3 Cooperative search approach

In order to improve the performance of the SS approach, we propose the *cooperative search approach* (CS), it used asynchronous communication among processors accessing the global state to eliminate the idle times. Each processor follows a separate search path, accesses the global state which consists of the current best solution and its cost whenever it finished a Markov subchain and updates the state if necessary. Once a processor gets the global state, it proceeds to the next Markov subchain with any delay.

Unlike SS, CS having the following characteristics:

- Idle times can be reduced since asynchronous communications overlap a part of the computation.
- Less communication overhead, an isolated access to the global state is needed by each processor at the end of each Markov subchain.
- The probability of being trapped in a local optimum can be smaller. This is because not all the processors start from the same state in each Markov subchain.

4. Experimental results

This section presents an experimental design and results derived from testing the parallel IS, SS, and CS algorithms described in Section 3. In order to show the performance of these approaches, two experiments were developed. The first experiment had as purpose to compare the three approaches in terms of parallel execution time. Among the three approaches, the CS approach was the fastest. The second experiment evaluated the quality solutions of CS approach over a new benchmark proposed in this paper. The results were compared against the best-known solutions reported in the literature [29] to construct covering arrays. The three parallel approaches were implemented using C

language and the message passing interface (MPI) library. The implementations were run on the *Tirant supercomputer*[1]. Tirant comprises 256 JS20 compute nodes (blades) and 5 p515 servers. Every blade has two IBM Power4 processors at 2.0 GHz running Linux operating system with 4 GB of memory RAM and 36 GB of local disk storage. All the servers provide a total of nearly 10 TB of disk storage accessible from every blade through GPFS (Global Parallel File System). Tirant has in total 512 processors, 1 TB of memory RAM and 19 TB of disk storage. The networks that interconnect the Tirant are:

1) Myrinet Network: High bandwidth network used by parallel applications communications.
2) Gigabit Network: Ethernet network used by the blades to mount remotely their root file system from the servers and the network over which GPFS works.

The following parameters were used for all SA implementations:

1) Initial temperature $T_i = 4.0$
2) Final temperature $T_f = 1.0E-10$
3) Cooling factor $\alpha = 0.99$
4) Maximum neighboring solutions per temperature $L = N \times k \times v^2$
5) Frozen factor $\phi = 11$
6) The neighborhood functions $\mathcal{N}_1$ and $\mathcal{N}_2$ are applied with a probability of $p_1 = 0.3$ and $p_2 = 0.7$ respectively.

4.1 Comparison of algorithms

To test the performance of the IS, SS, and CS approaches, we propose the construction of a covering array with $N = 80$, $t = 3$, $k = 22$, and $v = 3$. Each approach was executed 31 times (for provide statistical validity to experiment) using $P = \{4, 8, 16, 32\}$.

The performance of the algorithms has been compared based on median speed-up as a function of the number of processors, the results are shown in Fig. 2.

The IS approach, had difficulty in handling the large problem instances, it does not scale. The SS approach provides reasonable results, however, because it is a synchronous algorithm, the idle and communication times are inevitable. The CS approach is who offers the best results, it reduces the execution time of the SS approach by employing asynchronous information exchange.

In the next subsection, it is presented the second experiment of this work, the purpose is to measure the performance of the CS algorithm against the best-known solutions reported in the literature.

[1]The Tirant supercomputer: http://www.uv.es/siuv/cas/zcalculo/res/informa.wiki

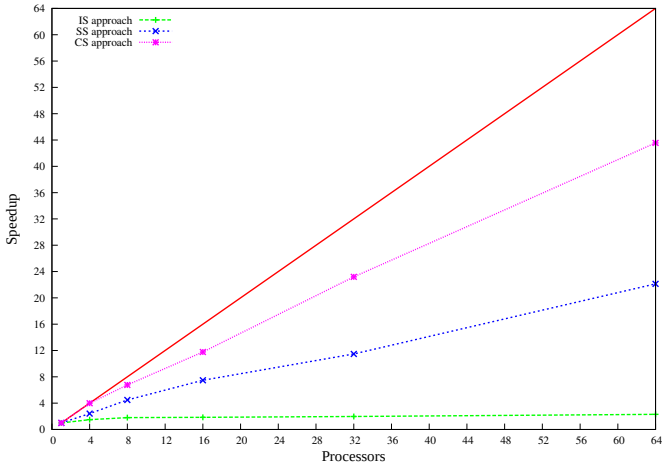


Fig. 2: Median speed-up as a function of the number of processors for the IS, SS, and CS approaches.

4.2 The cooperative search approach against the state-of-the-art procedures

The purpose of this experiment is to carry out a performance comparison of the bounds achieved by the CS approach with respect to the best-known solutions reported in the literature [29], which were produced using the following state-of-the-art procedures: orthogonal array construction, Roux type constructions, doubling constructions, algebraic constructions, Deterministic Density Algorithm (DDA), Tabu Search and IPOG-F.

For this experiment we have fixed the maximum computational time expended by our PSA for constructing a covering array to 72 hours and 50 processors. We create a new benchmark composed by 96 covering arrays with strength $t = 4$, degree $5 \leq k \leq 100$, and order $v = 3$. The detailed results produced by this experiment are listed in Table 1. The first two columns in this table indicate the strength t and the degree k of the selected instances. Next two columns show, in terms of the size N of the covering arrays, the best-known solution reported in the literature and the improved bounds produced by the CS.

The analysis of the data presented in the Table 1 lead us to the following observation. The solutions quality attained by the CS approach is very competitive with respect to that produced by the state-of-the-art procedures summarized in column 3 (ϑ). In fact, it is able to improve on 78 previous best-known solutions.

5. Conclusions

The long execution time of SA due to its sequential nature hinders its application to realistically sized problems, in this case, the CAC problem when $t > 3$, $5 \leq k \leq 100$, and $v = 3$. A more efficient way to reduce execution time and make the SA a more promising method is to parallelize

Table 1: Improved bounds on $CAN(4,k,3)$ for strength 4 and degrees $5 \leq k \leq 100$ produced by our PSA. ϑ represents the best-known solution reported in the literature [29]. β represents the best solution in terms of N produced by our PSA. Last column depicts the difference between the best result produced by our PSA and the best-known solution ($\Delta = \beta - \vartheta$).

(a)

t	k	ϑ	β	Δ
4	5	81	81	0
4	6	111	111	0
4	7	123	123	0
4	8	141	135	**-6**
4	9	159	135	**-24**
4	10	159	164	5
4	11	183	183	0
4	12	201	201	0
4	13	219	219	0
4	14	237	251	14
4	15	237	252	15
4	16	237	277	40
4	17	300	287	**-13**
4	18	307	300	**-7**
4	19	313	313	0
4	20	315	321	6
4	21	315	341	26
4	22	315	348	33
4	23	315	360	45
4	24	377	375	**-2**
4	25	384	375	**-9**
4	26	393	387	**-6**
4	27	393	387	**-6**
4	28	393	403	10
4	29	393	403	10
4	30	393	410	17
4	31	440	427	**-13**
4	32	445	432	**-13**
4	33	454	439	**-15**
4	34	462	447	**-15**
4	35	471	451	**-20**
4	36	471	459	**-12**

(b)

t	k	ϑ	β	Δ
4	37	471	461	**-10**
4	38	471	466	**-5**
4	39	471	468	**-3**
4	40	499	480	**-19**
4	41	506	484	**-22**
4	42	509	491	**-18**
4	43	518	495	**-23**
4	44	522	499	**-23**
4	45	526	505	**-21**
4	46	530	507	**-23**
4	47	534	511	**-23**
4	48	542	517	**-25**
4	49	549	523	**-26**
4	50	549	525	**-24**
4	51	549	531	**-18**
4	52	560	534	**-26**
4	53	565	537	**-28**
4	54	568	539	**-29**
4	55	572	545	**-27**
4	56	578	548	**-30**
4	57	581	553	**-28**
4	58	585	558	**-27**
4	59	589	560	**-29**
4	60	596	562	**-34**
4	61	596	567	**-29**
4	62	604	570	**-34**
4	63	604	574	**-30**
4	64	612	575	**-37**
4	65	617	581	**-36**
4	66	618	585	**-33**
4	67	623	587	**-36**
4	68	627	591	**-36**

(c)

t	k	ϑ	β	Δ
4	69	627	592	**-35**
4	70	627	597	**-30**
4	71	638	601	**-37**
4	72	639	601	**-38**
4	73	642	607	**-35**
4	74	645	607	**-38**
4	75	648	610	**-38**
4	76	653	613	**-40**
4	77	657	615	**-42**
4	78	657	618	**-39**
4	79	659	620	**-39**
4	80	663	624	**-39**
4	81	666	628	**-38**
4	82	668	631	**-37**
4	83	669	631	**-38**
4	84	669	635	**-34**
4	85	669	635	**-34**
4	86	669	639	**-30**
4	87	669	643	**-26**
4	88	669	643	**-26**
4	89	669	648	**-21**
4	90	669	649	**-20**
4	91	669	650	**-19**
4	92	669	653	**-16**
4	93	669	655	**-14**
4	94	669	657	**-12**
4	95	669	660	**-9**
4	96	669	661	**-8**
4	97	669	662	**-7**
4	98	669	664	**-5**
4	99	669	664	**-5**
4	100	669	665	**-4**

sequential SA. It is a challenging task. In fact, there are many approaches that may be considered in parallelizing SA. However, an inappropriate strategy used will likely result in poor performance.

In this paper, we have used three different approaches to do this work. From the experimental results, we found that the IS approach is the worst performing option, it does not scale. The SS approach offers reasonable execution times; compared to the IS, communication overhead in the SS approach would be increased. However, each processor can utilize the information from other processors such that the decrease in computations and idle times can be greater than the increase in communication overhead. For instance, a certain processor which is trapped in an inferior solution can recognize its state by comparing it with others and may accelerate the annealing procedure. That is, processors may collectively converge to a better solution. The CS approach is who offers the best results, it significantly reduces the execution time of the SS approach by employing asynchronous information exchange.

The quality of the solutions attained by the CS approach is very competitive with respect to that produced by the state-of-the-art procedures, in fact, it is able to improve on 78 previous best-known solutions and equals the solutions for other 7 instances.

Finally, the covering arrays are available in CINVESTAV Covering Array Repository (CAR), which is available under request at `http://www.tamps.cinvestav.mx/~jtj/CA.php`. We have verified all covering arrays described in this paper using the tool presented in [30].

Acknowledgments

The authors thankfully acknowledge the computer resources and assistance provided by Spanish Supercomputing Network (TIRANT-UV). This research work was partially funded by the following projects: CONACyT 58554, Calculo de Covering Arrays; 51623 Fondo Mixto CONACyT y Gobierno del Estado de Tamaulipas.

References

[1] A. Hartman, *Graph Theory, Combinatorics and Algorithms*, ser. Operations Research/Computer Science Interfaces Series. Springer US, 2005, vol. 34, ch. 10, pp. 237–266. [Online]. Available: http://dx.doi.org/10.1007/0-387-25036-0_10

[2] A. W. Williams and R. L. Probert, "A practical strategy for testing pair-wise coverage of network interfaces," in *Proceedings of the The Seventh International Symposium on Software Reliability Engineering - ISSRE 1996*. IEEE Computer Society, 1996, pp. 246–256. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.1996.558835

[3] G. B. Sherwood, "Optimal and near-optimal mixed covering arrays by column expansion," *Discrete Mathematics*, vol. 308, no. 24, pp. 6022 – 6035, 2008. [Online]. Available: http://dx.doi.org/10.1016/j.disc.2007.11.021

[4] L. Moura, J. Stardom, B. Stevens, and A. Williams, "Covering arrays with mixed alphabet sizes," *Journal of Combinatorial Designs*, vol. 11, no. 6, pp. 413–432, 2003. [Online]. Available: http://dx.doi.org/10.1002/jcd.10059

[5] S. S. Martirosyan and C. J. Colbourn, "Recursive constructions of covering arrays," *Bayreuther Mathematische Schriften*, vol. 74, pp. 266–275, 2005.

[6] C. J. Colbourn, S. S. Martirosyan, G. L. Mullen, D. Shasha, G. B. Sherwood, and J. L. Yucas, "Products of mixed covering arrays of strength two," *Journal of Combinatorial Designs*, vol. 12, no. 2, pp. 124–138, 2006. [Online]. Available: http://dx.doi.org/10.1002/jcd.20065

[7] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE Software*, vol. 13, no. 5, pp. 83–88, 1996. [Online]. Available: http://dx.doi.org/10.1109/52.536462

[8] Y. Tung and W. S. Aldiwan, "Automating test case generation for the new generation mission software system," in *Proceedings of the IEEE Aerospace Conference*, vol. 1. IEEE Press, 2000, pp. 431–437. [Online]. Available: http://dx.doi.org/10.1109/AERO.2000.879426

[9] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A general strategy for t-way software testing," in *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems - ECBS 2007*. IEEE Computer Society, 2007, pp. 549–556. [Online]. Available: http://dx.doi.org/10.1109/ECBS.2007.47

[10] R. C. Bryce and C. J. Colbourn, "The density algorithm for pairwise interaction testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 159–182, 2007. [Online]. Available: http://dx.doi.org/10.1002/stvr.365

[11] A. G. McDowell, "All-pairs testing," accessed on Feb 21, 2012. [Online]. Available: http://www.mcdowella.demon.co.uk/allPairs.html

[12] K. J. Nurmela and P. R. J. O. rd, "Constructing covering designs by simulated annealing," Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, Tech. Rep. B10, January 1993. [Online]. Available: ftp://saturn.hut.fi/pub/reports/B10.ps.Z

[13] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Augmenting simulated annealing to build interaction test suites," in *Proceedings of the 14th International Symposium on Software Reliability Engineering - ISSRE 2003*. IEEE Computer Society, 2003, pp. 394–405. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ISSRE.2003.1251061

[14] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete Applied Mathematics*, vol. 138, pp. 143–152, 2004. [Online]. Available: http://dx.doi.org/10.1016/S0166-218X(03)00291-9

[15] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01 - COMPSAC 2004*. IEEE Computer Society, 2004, pp. 72–77. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/CMPSAC.2004.1342808

[16] L. Gonzalez-Hernandez, N. Rangel-Valdez, and J. Torres-Jimenez, "Construction of mixed covering arrays of variable strength using a tabu search approach," in *Proceedings of the 4th international conference on Combinatorial optimization and applications - COCOA*, ser. Lecture Notes in Computer Science, vol. 6508. Springer-Verlag, 2010, pp. 51–64. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17458-2_6

[17] H. Avila-George, J. Torres-Jimenez, V. Hernández, and L. Gonzalez-Hernandez, "Simulated annealing for constructing mixed covering arrays," in *Proceedings of the 9th International Symposium on Distributed Computing and Artificial Intelligence - DCAI*, ser. Advances in Intelligent and Soft Computing, vol. 151. Salamanca, Spain, from 28th to 30th March: Springer Berlin / Heidelberg, 2012, pp. 657–664. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28765-7_79

[18] H. Avila-George, J. Torres-Jimenez, V. Hernández, and N. Rangel-Valdez, "A parallel algorithm for the verification of covering arrays," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA 2011*, 2011, pp. 879–885. [Online]. Available: http://world-comp.org/p2011/PDP8061.pdf

[19] E. H. L. Aarts and P. J. M. Van Laarhoven, "Statistical Cooling: A General Approach to Combinatorial Optimization Problems," *Philips Journal of Research*, vol. 40, pp. 193–226, 1985.

[20] D. J. Ram, T. H. Sreenivas, and K. G. Subramaniam, "Parallel simulated annealing algorithms." *Journal of Parallel and Distributed Computing*, vol. 37, no. 2, pp. 207–212, 1996. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1996.0121

[21] S.-Y. Lee and K. G. Lee, "Synchronous and asynchronous parallel simulated annealing with multiple markov chains," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 10, pp. 993–1008, 1996. [Online]. Available: http://dx.doi.org/10.1109/71.539732

[22] D.-J. Chen, C.-Y. Lee, C.-H. Park, and P. Mendes, "Parallelizing simulated annealing algorithms based on high-performance computer," *Journal of Global Optimization*, vol. 39, pp. 261–289, 2007. [Online]. Available: http://dx.doi.org/10.1007/s10898-007-9138-0

[23] Z. J. Czech, W. Mikanik, and R. Skinderowicz, "Implementing a parallel simulated annealing algorithm," in *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics - PPAM 2009*, ser. Lecture Notes in Computer Science, vol. 6067. Springer, 2010, pp. 146–155. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14390-8_16

[24] Y. Jun and S. Mizuta, "Detailed analysis of uphill moves in temperature parallel simulated annealing and enhancement of exchange probabilities," *Complex Systems*, vol. 15, no. 4, pp. 349–358, 2005. [Online]. Available: http://www.complex-systems.com/pdf/15-4-4.pdf

[25] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning," *Operations Research*, vol. 37, no. 6, pp. 865–892, 1989. [Online]. Available: http://www.jstor.org/stable/171470

[26] L. Cavique, C. Rego, and I. Themido, "Subgraph ejection chains and tabu search for the crew scheduling problem," *The Journal of the Operational Research Society*, vol. 50, no. 6, pp. 608–616, 1999. [Online]. Available: http://dx.doi.org/10.1057/palgrave.jors.2600728

[27] M. Atiqullah, "An efficient simple cooling schedule for simulated annealing," in *Proceedings of the International Conference on Computational Science and its Applications - ICCSA 2004*, ser. Lecture Notes in Computer Science, vol. 3045. Springer-Verlag, 2004, pp. 396–404. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24767-8_41

[28] Z. Czech, "Three parallel algorithms for simulated annealing," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, vol. 2328. Springer Berlin / Heidelberg, 2006, pp. 210–217. [Online]. Available: http://dx.doi.org/10.1007/3-540-48086-2_23

[29] C. J. Colbourn, "Covering array tables for t=2,3,4,5,6," accessed on March 16, 2012. [Online]. Available: http://www.public.asu.edu/~ccolbou/src/tabby/catable.html

[30] H. Avila-George, J. Torres-Jimenez, N. Rangel-Valdez, A. Carrión, and V. Hernández, "Supercomputing and Grid computing on the verification of covering arrays," *The Journal of supercomputing*, pp. 1–30, 2012, published online: 18 April 2012. [Online]. Available: http://dx.doi.org/10.1007/s11227-012-0763-0

Exact and Approximate Median Splitting on Distributed Memory Machines

Matthieu Garrigues and Antoine Manzanera
UEI, ENSTA-ParisTech, Paris, France

Abstract— *We present in this paper a new fine grained median split algorithm which places the median on the middle index of an input array of size N. Running on P processors, the randomized version converges in $O\left(\frac{N}{P} \times (log(N) + \mu)\right)$ average time where $0 < P < \frac{N}{4}$ and μ is the time to swap a pair of elements between two nodes. At each iteration, the nodes process only its local elements and exchange part of them with another processor of the network. This makes it a decentralized parallel algorithm and offers the possibility to take advantage of massively parallel computing networks based on distributed memory systems.*

Keywords: median splitting, parallel algorithms, distributed memory machines

1. Introduction

Useful in many fields like statistics or signal processing median selection has been widely used in computer science in the last decades. Given a set of N elements from a totally ordered space, it consists in picking the element that is superior and inferior to the same number of elements. A expensive way of solving the problem would be to sort the whole input so that the median index is $\lceil \frac{N}{2} \rceil$.

Several non parallel algorithms exist. Given a pivot value p, quickSelect [1] splits the input array in two parts, places inferior elements before p and superior elements after, then recursively processes the part that contains the median. In average, it divides the problem size by a factor two after each iteration. Its average run-time is in $O(N)$ but at worst in $O(N^2)$. Median of median [1] provides a method for choosing a pivot that ensures that the run-time stays in $O(N)$ but is less efficient in practice. [2] is a randomized algorithm that successively estimates an interval that contains the median value.

Many works present algorithms that efficiently run on different PRAM models. [3] shows how median selection can map on coarse-grained computers, when the input size is an order of magnitude higher than the number of processors. It presents and benchmarks parallel implementations of [1] and [2] on the connection machine 5 (CM-5). In theses approaches, which reduce the number of elements to process after each iterations, load balancing is needed in order to feed all processors at any iteration. Randomized parallel algorithms in [4] and [5] require $O(loglog(N))$ steps for convergence. In [6], the author presents some fine grained and real-time hardware implementation of median filtering.

In this paper, we present a new fine grained distributed median randomized algorithm. In opposition to the state of the art algorithms, it is efficient on networks (See Fig. 2) containing $O(N)$ (up to $\frac{N}{4} - 1$) processing units running in parallel. Its average observed run-time is $O\left(\frac{N}{P} \times (log(N) + \mu)\right)$ where N is the input size, P the number of processors and μ the time required to swap 2 elements between two nodes. This paper is organized as follows: We first introduce the basic operators used as building blocks. Then, we present three algorithms in their sequential form, from the simpler to the most efficient one. Finally, we discuss about the parallel aspects of the final method in Sect. 5 and its convergence in Sect. 6.

2. Primitives and Notations

Let A be an array of elements drawn from some totally ordered set. $\forall i \in \{0, N-1\}, A[i]$ represents the ith element of the array.

The algorithms presented in this paper make use of the following primitives:

- $reorder(A, i, j)$ (and by extension $reorder(A, i, j, k)$) reorders the ith and jth (resp. the ith, jth, and kth) elements of A, such that $A[i] \leq A[j]$ (resp. $A[i] \leq A[j] \leq A[k]$) after the call.
- $select_{\min}(A, i, j, k)$ (resp. $select_{\max}(A, i, j, k)$) exchanges, if smaller (resp. greater), $A[i]$ with $\min(A[j], A[k])$ (resp. $\max(A[j], A[k])$).

Each instance of these operations is said *effective* and returns true if it actually modifies the input array. If not, it returns false.

3. Swap on Tree

In this section we present the basic version of our algorithm: the principle is to map the input vector A onto a binary tree, reorganizing the data such that the left half subtree is a inf-semilattice, the right half subtree is a sup-semilattice, and the median is on the root. See Fig. 1 for a graphical representation, showing the left (resp. right) part of the tree under (resp. over) the root.

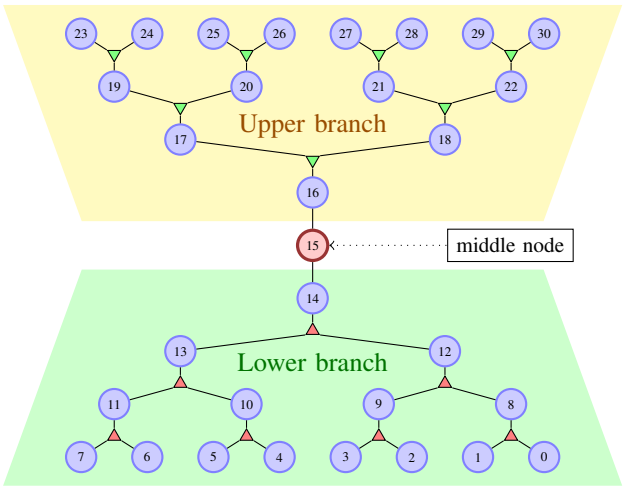


Fig. 1: A vector of 31 elements mapped on a binary tree made of an inf-semilattice (upper branch), a sup-semilattice (lower branch) and a middle node (median). Each node is labelled with its position in the array. The triangles represent the $select_{\min}$ and $select_{\max}$ operators.

The algorithm updates the input A using $select_{\min}$ (in the upper branch), $select_{\max}$ (in the lower branch), and $reorder$ (at the middle node). For simplicity we suppose that dimension $N = 2^k - 1, k > 1$.

Listing 1: Algorithm swap on tree

```
1 middle = (N-1)/2
2 repeat {
3   for (i = (middle - 1)/2; i ≥ 1; i--)
4     //lower branch
5     selectmax(A, middle-i, middle-2i, middle
          -2i-1)
6     //upper branch
7     selectmin(A, middle+i, middle+2i, middle
          +2i+1)
8   endfor
9   //middle node
10  eff = reorder(A, middle-1, middle, middle
        +1)
11 } until not eff
```

a) Complexity Analysis: Every iteration exchanges the greatest element of the lower branch with the smallest of the upper branch. Convergence occurs as soon as the *reorder* instruction (line 10) becomes uneffective, which takes at most $N/2$ iterations.

Considering the binary tree, the inputs of even levels are the output of odd levels (and *vice versa*). Then, half the tree can be processed in parallel, using $N/4$ concurrent tasks. However the number of iterations remains in $O(N)$, then the run time is $O\left(\frac{N^2}{P}\right)$ using P processors. The poor efficiency of this basic version comes from the fact that all the elements that are not placed in the good half of the array (at most $N/2$ in every half) will have to pass through the middle node, because only one element can travel from the upper branch to the lower branch at each iteration. We address this bottleneck issue in the next section.

4. Swap Inter Branches

We describe here an algorithm that improves the *connectivity* between the lower and upper branches by providing shortcuts to travel from one branch to the other.

Keeping the instructions of the *swap on tree* version, we also reorder each node of the lower branch with its counterpart in the upper branch; more precisely, $\forall i \in [0, \lfloor N/2 \rfloor - 1]$, $A[i]$ and $A[N - i - 1]$ are reordered. In a nutshell, the $select$ operators guarantees the convergence, whereas $reorder$ works as a catalyst.

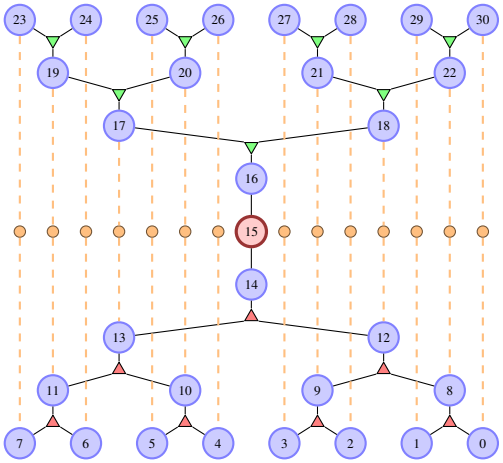


Fig. 2: Binary tree drawn on a vector of 31 elements. The circles and dashed lines show the additional $reorder$ operators of the algorithm *swap inter branches*. Each node is labelled with its position in the array.

Note that elements in the upper branch are only reordered with elements of the same level in the lower branch. The method takes advantage of the fact that at iteration i the probability of an element in the level l to be in the wrong branch is higher if l is close to the middle node. This property becomes stronger when i increases.

a) Complexity Analysis: This version keeps the high potential of parallelism, since all the instructions in the *swap inter branches* loop can be executed concurrently on $N/2$ processors.

After one iteration of *swap inter branches* instructions (line 4), there remain at most $N/4$ misplaced elements in every branch (otherwise it would mean that $k > N/4$

Listing 2: Algorithm swap inter branches

```
1 middle = (N-1)/2
2 repeat {
3   for (i = 0; i < middle; i++)//swap inter
        branches
4     reorder(A,i,N-i-1)
5   endfor
6   for (i = (middle - 1)/2; i ≥ 1; i--)
7     //lower branch
8     selectmax(A, middle-i, middle-2i, middle
          -2i-1)

9     //upper branch
10    selectmin(A, middle+i, middle+2i, middle
          +2i+1)

11  endfor
12  //middle node
13  eff = reorder(A, middle-1, middle, middle
        +1)
14 } until not eff
```

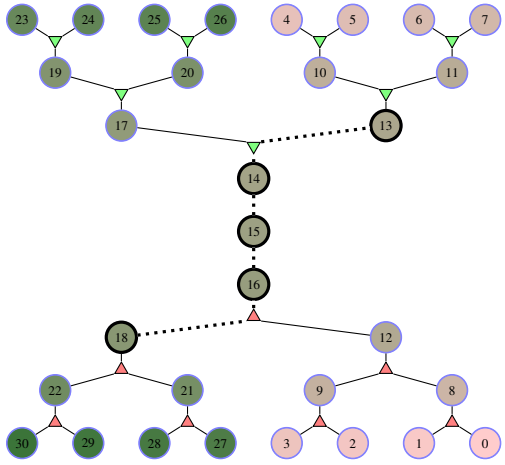


Fig. 3: Worst case for *swap inter branches*. the label and the color of a node represent the ranks of the value that it stores. The bottleneck is highlighted in dashed black.

elements smaller than the median are smaller than k other elements in the other branch and then there would be $2k > N/2$ elements smaller than the median, which is absurd). Unfortunately, the number of iterations is still $N/4$ in the worst case (see Fig. 3), so only two times faster than the *swap on tree* version. It turns out that this version still suffers from a bottleneck: in the *swap inter branches* instructions, every node is always reordered with its vertical counterpart in the other branch. Then, if some elements belonging to two diagonally opposed quarters of the tree need to be compared (like in Fig. 3), they still have to go through the middle node. Furthermore, the $select_{\min}$ and $select_{\max}$ operations (lines 8 and 10) often perform the same comparisons, since many elements do not move during one iteration (this is especially true during the last iterations). Finally, a lot of operations are not effective. In the next section we present the final version of the algorithm, adding horizontal motion at each level to improve the data mobility.

5. Final Algorithm

This section presents our final algorithm, which improves the previous version by changing at every iteration the counterpart of every node, thanks to index shifting at each level of the tree. The purpose is to minimize the bottleneck effect by allowing diagonal motion in the *swap inter branches* instructions, and to maximize the efficiency of the $select_{\min}$ and $select_{\max}$ operations, by a constant renewal of the data caused by the horizontal motion (see Fig. 4). These improvements allow to converge in $O(log(N))$ steps instead of $O(N)$ with the previous algorithm. We present the sequential version (List. 3), where RANDOM(N) is a random integer between 0 and N-1, and MOD(p,n) is the remainder of the Euclidean division of p by n, and then discuss about parallelization.

Listing 3: Final algorithm, sequential form

```
1 middle = (N-1)/2
2 repeat {
3   eff = false
4   //size of the leaf level
5   Lsize = (N+1)/4
6   //first index of the leaf level (lower
        branch)
7   Plow = Lsize-1
8   //first index of the leaf level (upper
        branch)
9   Pupp = N-Lsize
10  //shift offset of the leaf level (lower
        branch)
11  Olow_son = RANDOM(Lsize)
12  //shift offset of the leaf level (upper
        branch)
13  Oupp_son = RANDOM(Lsize)
14
15  repeat {//Processing of one level
16    Olow_dad = RANDOM(Lsize/2)
17    Oupp_dad = RANDOM(Lsize/2)
18    for (k = 0; k < Lsize/2; k++)
19      Ilow_dad = Plow + Lsize/2
20                 - MOD(k + Olow_dad, Lsize/2)
21      Iupp_dad = Pupp - Lsize/2
22                 + MOD(k + Oupp_dad, Lsize/2)
23      Ilow_child1 = Plow - MOD(2k + Olow_son,
            Lsize)
24      Ilow_child2 = Plow
25                    - MOD(2k + 1 + Olow_son,
                          Lsize)
26      Iupp_child1 = Pupp + MOD(2k + Oupp_son,
            Lsize)
27      Iupp_child2 = Pupp
28                    + MOD(2k + 1 + Oupp_son,
                          Lsize)
29      //lower branch
```

```
30        selectmax(A, Ilow_dad, Ilow_child1,
              Ilow_child2)
31        //upper branch
32        selectmin(A, Iupp_dad, Iupp_child1,
              Iupp_child2)
33        //swap inter branches
34        eff = eff || reorder(A, Ilow_dad,
              Iupp_dad)

35      endfor
36      Plow = Plow + Lsize/2
37      Pupp = Pupp - Lsize/2
38      Olow_son = Olow_dad
39      Oupp_son = Oupp_dad
40      Lsize = Lsize/2
41    } until (Lsize == 1)
42    //middle node
43    reorder(A, middle - 1, middle, middle + 1)
44  } until not eff
```

At each iteration, we scan the nodes from the leaves to the middle node using $reorder$, $select_{\max}$ and $select_{\min}$. A $select$ operation rearranges one element of a pair of nodes at level l with its parent node at level $l-1$. But, unlike *swap inter branches*, relations between levels are shifted with a random offset drawn before each iteration. Figure 4 shows the resulting relations. Arguments of $reorder$ are affected using the same method.

a) Convergence detection: The variable eff tracks whether during one iteration a $reorder$ between branches has been effective (see line 34 of listing 3). If not, we can ensure that the median is on the middle node.

b) Parallelization: At a given time, one iteration reads and writes two opposite children branch levels, and two opposite father branch levels. Since each iteration processes nodes from the leaves to the middle of the tree, we can pipeline the iterations without breaking write/read dependencies. Thus, using a pipeline of $log_2(N)/2$ stages, as many iterations can run in parallel (see Fig. 5).

c) Complexity Analysis: If we suppose that the random number generator used to compute offset always returns 0, the algorithm is equivalent to *swap and shift*. In this case, the number of iterations in the worst case is still $N/4$. Section 6 shows that this algorithm takes in average $2 \times log_2(N)$ pipeline steps to converge. Using P processors, one pipeline step runs in time $O(\frac{N}{P})$. This gives an average time complexity of $O\left(log(N) \times \frac{N}{P}\right)$. Figure 6 compares the convergence of the three presented algorithms.

6. Convergence

Let us consider the input array being re-arranged at a certain step of the algorithm. We define the event e as follows: an element drawn at random from the array is misplaced (i.e. inferior (resp. superior) to the median and located in the upper (resp. lower) branch). In this section we model the evolution of the probability $P(e)$ over the pipeline steps. First, we analyze how $P(e)$ is affected by the operators $select_{\min}$, $select_{\max}$ and $reorder$. Then we observe the average number of pipeline steps needed to place all elements.

Let a and b be two consecutive levels, a carrying the parents of nodes on level b. The event e_x (resp. f_x) occurs

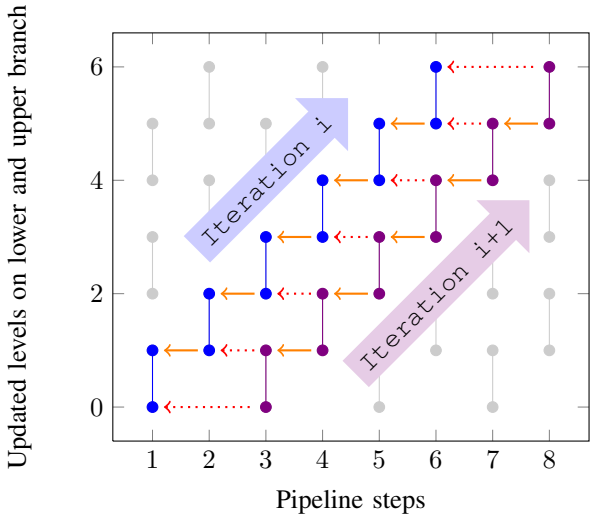


Fig. 5: Dependencies between steps of two iterations. Plain arrows show dependencies between the steps of one iteration, dependencies between iterations are dotted.

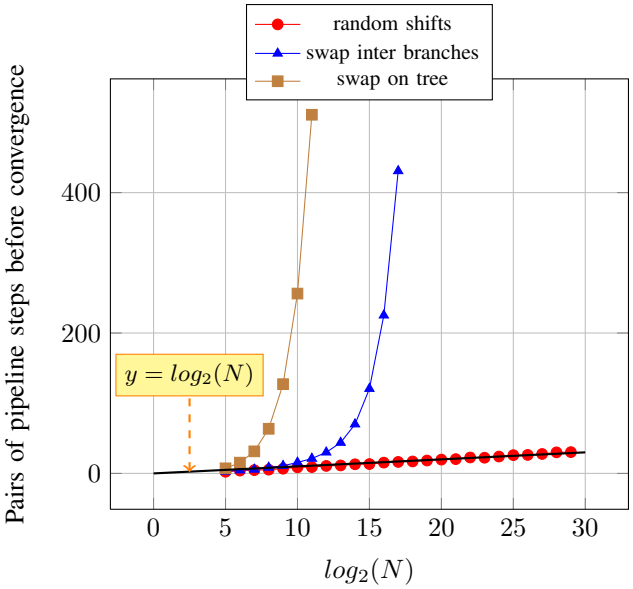


Fig. 6: Average pairs of pipeline steps before convergence on different algorithms. Means of 100 executions on random inputs.

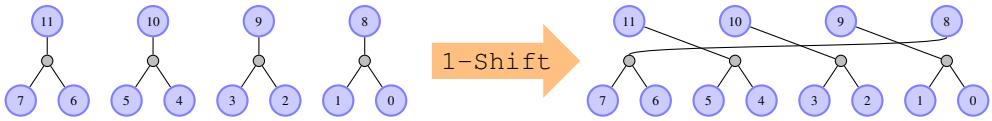


Fig. 4: Example of index shifting using an offset of size 1. Unmodified (left) and shifted (right) relations between two levels. *select* operators are shown in gray. Each nodes is labeled with its position in the array.

when an element is misplaced on level x before (resp. after) the *select* operation. On each branch, level 0 contains the root and level l contains the parents of level $l+1$. Given $p_x = P(e_x)$, we search S_p and S_l such that:

$$\begin{cases} P(f_a) & = S_p(p_a, p_b) \\ P(f_b) & = S_l(p_a, p_b) \end{cases}$$

Figure 7 shows all the possible configurations for the three operators.

$$\begin{aligned} S_p(p_a, p_b) &= \sum_{i=1}^{8} P(f_a|K_i) \times P(K_i) \\ &= 1 - P(\overline{f_a}|K_1) \times (1-p_a) \times (1-p_b)^2 \\ &= 1 - (1-p_a) \times (1-p_b)^2 \end{aligned}$$

$$\begin{aligned} S_l(p_a, p_b) &= \sum_{i=1}^{8} P(f_b|K_i) \times P(K_i) \\ &= P(f_b|K_4) \times (1-p_a) \times p_b^2 + P(f_b|K_6) \times \\ &\quad p_a \times (1-p_b) \times p_b + P(f_b|K_8) \times p_a \times p_b^2 \\ &= \frac{(1-p_a) \times p_b^2}{2} + p_a \times (1-p_b) \times p_b + p_a \times p_b^2 \\ &= p_b \times \left(\frac{p_b \times (1-p_a)}{2} + p_a \right) \end{aligned}$$

Using the same method, we model how *reorder* affects $P(e)$ on level l of lower and upper branches. The event e (resp. f) occurs when an element is misplaced on level l before (resp. after) the operation. The probability of e is noted $p = P(e)$. R defines the relation between p and $P(f)$: $P(f) = R(p)$.

$$\begin{aligned} R(p) &= \sum_{i=1}^{4} P(f|L_i) \times P(L_i) \\ &= P(f|L_1) \times (1-p)^2 + \\ &\quad \big(P(f|L_2) + P(f|L_3)\big) \times p \times (1-p) + P(f|L_4) \times p^2 \\ &= (1-p) \times p \end{aligned}$$

Let e_l^s denote the following event: an element of level l is misplaced after running the step s. It occurs with a probability $P_l^s = P(e_l^s)$. Knowing that a step is a *select* followed by a *reorder* we can build P_l^s by recurrence on s:

$$P_l^s = \begin{cases} R(S_p(P_l^{s-1}, P_{l+1}^{s-1})) & \text{if} \quad l \bmod 2 = s \bmod 2 \\ S_l(P_{l-1}^{s-1}, P_l^{s-1}) & \text{otherwise} \end{cases}$$

P_l^0 is given by the distribution of the input array. In case of all orderings of the input are equally likely, we have: $\forall l,\ P_l^0 = \frac{1}{2}$. On the root and leaf levels of each branch, the previous formula is undefined every two pipeline steps. In this special case, we have $P_l^s = P_l^{s-1}$. Figure 8 draws the convergence of P_l^s and the probability P^s that an element of the input is misplaced after step s:

$$P^s = \sum_{i=1}^{log_2(N+1)-1} \frac{2^i \times P_l^s}{N}$$

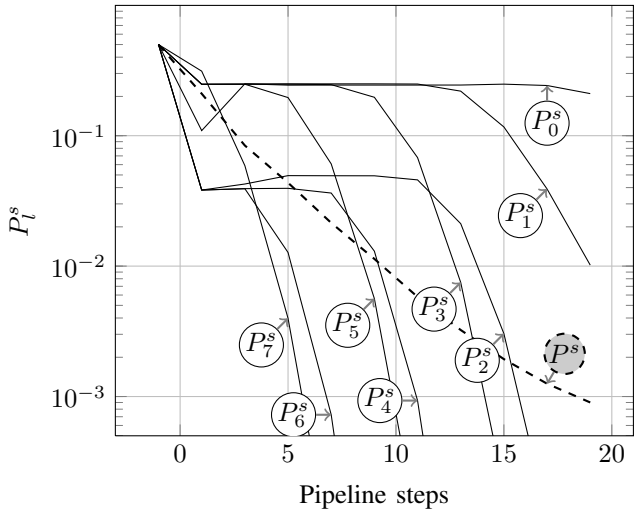


Fig. 8: Evolution of P_l^s and P^s for a input of 2^9-1 elements with $P_l^0 = \frac{1}{2}\ \forall l$. P^s shows that in average, the number of placed elements is divided by two after each pair of pipeline steps. The larger levels are the first to contain placed elements with high probability. For clarity, only values with s odd are displayed.

a) Approximation: Figure 8 shows that the probability for an element to be misplaced is reduced by a factor two every two pipeline steps. If we stop the algorithm after a given step s, we can estimate, for each level, the probability P_l^s that

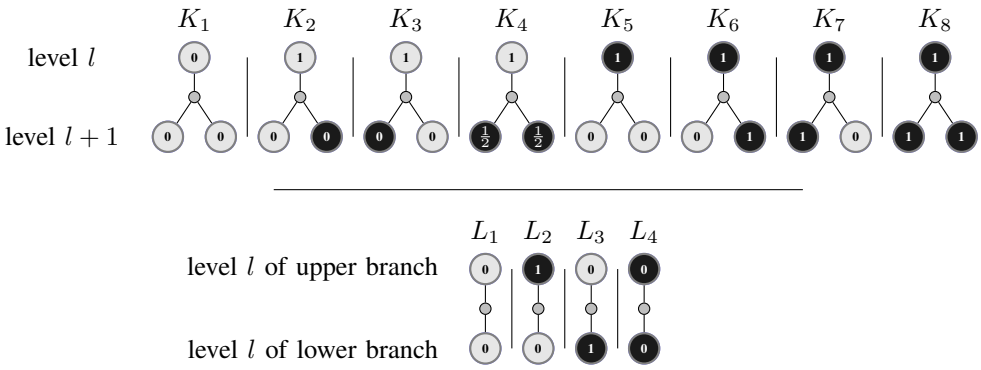


Fig. 7: All possible configurations for *reorder* and *select* operators. Misplaced elements are shown in black and placed elements are shown in gray. Each node is marked with the probability that its contains a misplaced element after the operation.

one of its elements is misplaced. Figure 9 shows the average percentage of misplaced elements after each pair of pipeline steps. For example, whatever the input size, we note that 10 pipeline steps are enough to place 99% of the elements in average.

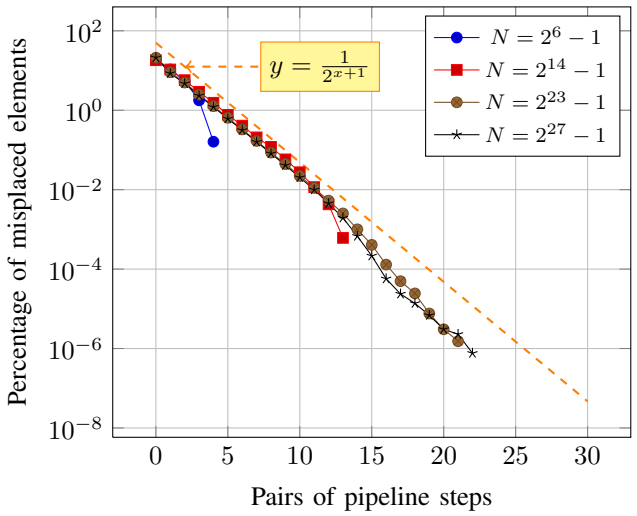


Fig. 9: Evolution of the percentage of misplaced elements over the pipeline steps. Each curve represents the mean of 10 executions of the final algorithm on random arrays.

7. Conclusion

The algorithm we presented in this paper converges to a median split of an array of N elements, placing the median at index $\frac{N}{2}$ in an observed run-time of $O\left(\frac{N}{P} \times (log(N) + \mu)\right)$ average time where μ the time to swap a pair of elements between two processors.

To make the coarse-grained parallel approaches of [3] efficient, we need to provide each processing units (PE) with enough elements, limiting their number. Because the three operators involved process locally only 2 or 3 elements without global knowledge, our approach is expected to bypass this limitation by taking advantage of architectures with a very high number of PEs (in the same order of magnitude of N). We believe that this can lead to efficient implementations on specific networks on chip with thousands of PEs.

References

[1] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *J. Comput. Syst. Sci.*, vol. 7, pp. 448–461, August 1973. [Online]. Available: http://dx.doi.org/10.1016/S0022-0000(73)80033-9

[2] R. W. Floyd and R. L. Rivest, "Expected time bounds for selection," *Commun. ACM*, vol. 18, pp. 165–172, March 1975. [Online]. Available: http://doi.acm.org/10.1145/360680.360691

[3] I. Al-furiah, S. Aluru, S. Goil, and S. Ranka, "Practical algorithms for selection on coarse-grained parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, pp. 813–824, 1997.

[4] S. Rajasekaran, "Randomized selection on the hypercube," *J. Parallel Distrib. Comput.*, vol. 37, no. 2, pp. 187–193, Sept. 1996. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1996.0118

[5] D. A. Bader, "An improved, randomized algorithm for parallel selection with an experimental study," *J. Parallel Distrib. Comput.*, vol. 64, no. 9, pp. 1051–1059, Sept. 2004. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2004.06.010

[6] R. M. Palenichka, "Parallel median filtering algorithms and their real-time implementation," *Cybernetics and Systems Analysis*, vol. 25, pp. 694–699, 1989, 10.1007/BF01075231. [Online]. Available: http://dx.doi.org/10.1007/BF01075231

SMC-PBC-SVM: A Parallel Pmplementation of Support Vector Machines for Data Classification

Rabie Ahmed[§], Adel Ali, Chaoyang Zhang[*]
School of computing, University of Southern Mississippi, Hattiesburg, USA
Rabie.Ahmed@eagles.usm.edu, Adel.Ali@usm.edu, Chaoyang.Zhang@usm.edu

Abstract- The **Support Vector Machine (SVM) is one of the most effective machine learning algorithms for data classification, which have a significant area of research. Since the training process of large datasets is computationally intensive, there is a need to improve its efficiency using high performance computing techniques. In this paper, we developed an efficient parallel algorithm, SMC-PBC-SVM, which combines a Parallel Binary Class with Serial Multi-class Support Vector Machines for classification. The SMC-PBC-SVM algorithm was implemented using the object-oriented C++ programming language and standard Message passing Interface (MPI) communication routines. The parallel code was executed on an ALBACORE Linux cluster, and then tested with four datasets with different sizes: Earthworm, Protein, Mnist, and Mnist8m. The results show that the SMC-PBC-SVM implementation can significantly improve the performance of data classification without the loss of accuracy. The results also demonstrated a form of proportionality between the size of the dataset and the SMC-PBC-SVM efficiency. As the dataset becomes larger, the SMC-PBC-SVM achieves a higher efficiency.**

Keywords- Classification; SVM; parallel computing.

I. INTRODUCTION

Classifying different categories among large datasets has become one of the most important computing problems. The main objective in classification is to identify patterns in a data set, which helps to analyze the data in order to make decisions. Support Vector Machines (SVMs) are a class of machine learning algorithms based on statistical learning theory, which has received wide attention for classification problems because of its accuracy and generalization property.

SVM classification involves three stages. The first one involves training the model for the classification with the training dataset. The second stage is the testing stage where the model is tested with a combination of the training data and similar unseen data. The third stage involves the actual prediction with unseen data. The training stage is the most computationally expensive process of SVMs.

The main idea behind the SVM classification algorithm is to separate two point classes of a training dataset,

$$D = \{(x_i, y_i), i = 1,2, \dots, n, \quad x_i \in R^M, \quad y_i \in \{-1,1\}\} \quad (1)$$

, with a surface that maximizes the margin between them [1]. This separating surface is obtained by solving a convex quadratic problem (QP) of the form [2]

$$\text{Min } F(\alpha) = \frac{1}{2}\alpha^T G \alpha - \sum_{i=1}^{n} \alpha_i \ , \quad (2)$$

$$\text{sub. to } \sum_{i=1}^{n} \alpha_i \, , 0 \le \alpha_i \le C, i = 1,2, \dots, n$$

, where the entries of the symmetric positive semi-definite matrix G are defined as

$$G_{ij} = y_i y_j K(x_i, x_j), i, j = 1,2, \dots, n, \quad (3)$$

where $K: R^M \times R^M \to R$ is the kernel function.

SVM has been modified to handle non-linear classification. Since the complexity of training of non-linear SVMs has been estimated to be quadratic in the number of training examples [3], it is computationally expensive when large datasets with tens of thousands of training examples are used. To reduce the training time, the optimization problem can be broken into smaller QP problems [4]. Originally, SVM was introduced for binary classification, and then it was extended for multi-class classification. It was improved by caching the kernel calculations [5]. Because of the wide use of the Internet, a large amount of data is being collected. Hence, the importance of using an efficient SVM that utilizes parallel computing facilities and multi-core processing elements for (multi-class) classification of large datasets grows even larger. Therefore, a lot of research efforts were directed to find the optimal parallel algorithm for the different kinds of datasets. For large binary classification problems, there is a need to break it down into smaller pieces, so that the smaller partitions can be computed concurrently. Research has been conducted in this area, and some progress has been made in [6], [7], and [8]. On the other hand, for large multi-class classification problems, progress has been made in [3] and [9]. Also, a lot of work has been done in [5] and [10] to develop kernel computation costs. Some other efforts have been achieved in [11], [12], [13] and [14] to optimize working set size selection. Additionally, other tries have been done in [15], and [16] to develop SVM training by quickly removing most of non-support vectors.

The LIBSVM [17] software is developed for a working set of size two, which tends to minimize the computational cost per iteration. In this case, the inner QP subproblem can be systematically solved without requiring a numerical QP solver and the updating of the objective gradient only involves the two Hessian columns corresponding to the two updated

*Correspondence: USM, School of Computing, Chaoyang.Zhang@usm.edu.
§Permanent address: Beni Suef University, Faculty of Science, Beni Suef, Egypt.

variables. On the other hand, if few variables are updated per iteration, slow convergence is normally implied. The SVM^{light} [18] algorithm uses a more general decomposition strategy, also by common sense it can exploit working sets of sizes larger than two. By updating more variables per iteration, such an approach is more suitable for a faster convergence, but it introduces additional difficulties and costs. A generalized maximal-violating pair strategy for the working set selection and a numerical solver for the inner QP subproblems are required. Moreover, as more variables are updated per iteration, the objective gradient updating is more expensive. While SVM^{light} can run with any working set size, numerical experiences prove that it effectively faces the above difficulties only in the case of small sized working sets, $N_{sp} = O(10)$, where it often exhibits comparable performance with LIBSVM.

Following the SVM^{light} decomposition techniques, another effort to strike a balance between the convergence rate and cost per iteration was introduced in [6]. Unlike SVM^{light}, it is medium or large sized working sets, ($N_{sp} = O(10^2)$ or $N_{sp} = O(10^3)$, that allow the method to converge in a small number of iterations where the most costly tasks are. For example, subproblem solving and gradient updating can be simply and effectively distributed between the available processors. Based on this idea, a new parallel gradient projection-based decomposition technique (PGPDT) is developed and implemented in software to train support vector machines for binary classification problems in parallel as in [8].

Even though all previous results were encouraging, more research was needed to improve the performance of the process. In this paper, we introduce a new classification algorithm which merges parallel binary classification with serial multi-class classification to produce an efficient parallel algorithm for classification. We named the new algorithm SMC-PBC-SVM.

II. SMC-PBC-SVM ALGORITHM

The SMC-PBC-SVM algorithm combines Parallel Binary Class with Serial Multi Class Support Vector Machines for classification. It includes seven steps and works as follows: 1) Reads a dataset from an input file, 2) Groups samples of the same class together, 3) Collects each two classes into one task, 4) Sorts $\frac{K(K-1)}{2}$ tasks based on its size, 5) Divides processes group into two subgroups, mulgroup and bingroup, such that mulgroup is used to build $\frac{K(K-1)}{2}$ binary tasks where K is the number of classes in the dataset, and bingroup is used to solve each binary task from $\frac{K(K-1)}{2}$ tasks in parallel, 6) Builds SVM model after solving all binary tasks, and 7) Writes the SVM model into an output model file which is used to predict testing dataset file. The algorithm flowchart is illustrated in Figure 1.

Since the SMC-PBC-SVM is based on the Parallel Binary Class algorithm implemented by PGPDT, we briefly explain here the fundamental principles and decomposition technique used in [8]. We start that by stating some fundamental principles. At each decomposition iteration, the indices of the variables $\alpha_i, i = 1,2,..,n$, are split into the set B of basic variables, usually called the working set, and the set N = {1,2,…,n} \ B of nonbasic variables. In consequence, the kernel matrix G, the vectors $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n)^T$, and the vector $y = (y_1, y_2, \dots, y_n)^T$ can be arranged with respect to B and N as follows:

$$G = \begin{bmatrix} G_{BB} & G_{BN} \\ G_{NB} & G_{NN} \end{bmatrix}, \qquad \alpha = \begin{bmatrix} \alpha_B \\ \alpha_N \end{bmatrix}, \qquad y = \begin{bmatrix} y_B \\ y_N \end{bmatrix}$$

Then, suppose that N_{sp} is the size of the working set, where $N_{sp} = \#B$ and α^* is a solution of QP (2), and N_p is the number of processes which are used for solving that QP, and each of them has a local copy of the training set D (1), where the entries of G are defined by G(3). The decomposition technique used by the PGPDT falls within the general idea stated in PGPDT algorithm, which is shown in Figure 2. Label "Distributed task" in A2 and A3 of PGPDT algorithm refers to the steps where the N_p processors cooperate together to perform the required computation. In these steps, communications and synchronization are needed. In the other steps, the processors asynchronously perform the same computations on the same input data to obtain a local copy of the expected output data.

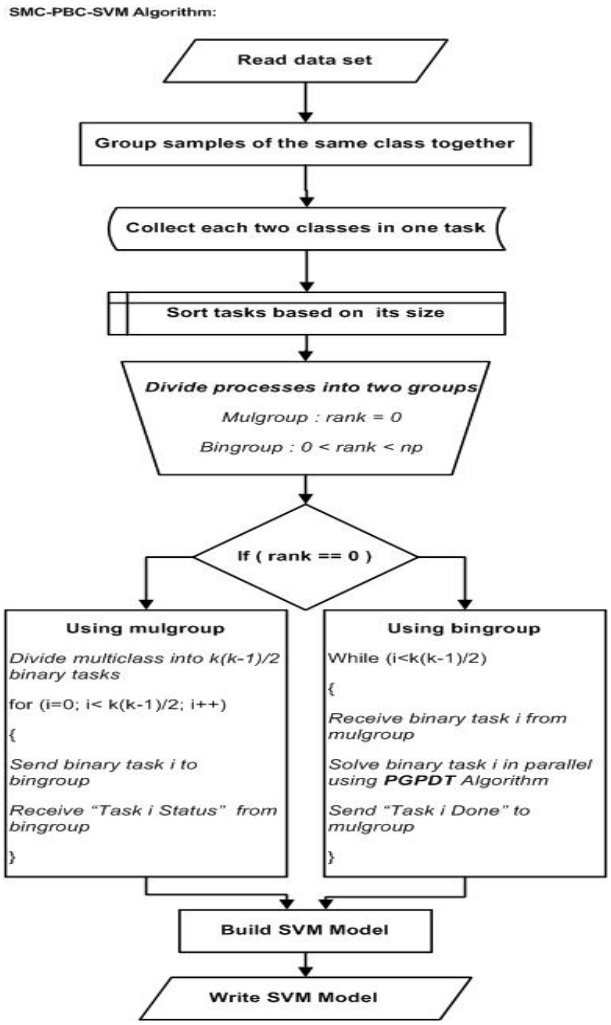


Figure 1. Serial Multi Class Parallel Binary Class Support Vector Machines

A1. *Initialization.* Set $\alpha^{(1)} = 0$ and let n_{sp} and n_c be two integer values such that $n \geq n_{sp} \geq n_c$ n_c even. Choose n_{sp} indices for the working set $\mathcal{B}$ and set $k = 1$.

A2. *QP subproblem solution.* **[Distributed task]** Compute the solution $\alpha_{\mathcal{B}}^{(k+1)}$ of

$$\begin{aligned} \min \quad & \frac{1}{2}\alpha_{\mathcal{B}}^T G_{\mathcal{B}\mathcal{B}}\alpha_{\mathcal{B}} + \left(G_{\mathcal{B}\mathcal{N}}\alpha_{\mathcal{N}}^{(k)} - 1_{\mathcal{B}}\right)^T \alpha_{\mathcal{B}} \\ \text{sub. to} \quad & \textstyle\sum_{i\in\mathcal{B}} y_i\alpha_i = -\sum_{i\in\mathcal{N}} y_i\alpha_i^{(k)}, \\ & 0 \leq \alpha_i \leq C, \quad \forall i \in \mathcal{B}, \end{aligned}$$

where $1_{\mathcal{B}}$ is the n_{sp}-vector of all one; set $\alpha^{(k+1)} = \left(\alpha_{\mathcal{B}}^{(k+1)^T}, \ \alpha_{\mathcal{N}}^{(k)^T} \right)^T$.

A3. *Gradient updating.* **[Distributed task]** Update the gradient

$$\nabla f\left(\alpha^{(k+1)}\right) = \nabla f\left(\alpha^{(k)}\right) + \begin{bmatrix} G_{\mathcal{B}\mathcal{B}} \\ G_{\mathcal{N}\mathcal{B}} \end{bmatrix} \left(\alpha_{\mathcal{B}}^{(k+1)} - \alpha_{\mathcal{B}}^{(k)}\right)$$

and terminate if $\alpha^{(k+1)}$ satisfies the KKT conditions.

A4. *Working set updating.* Update $\mathcal{B}$ by the following selection rule:

A4.1. Find the indices corresponding to the nonzero components of the solution of

$$\begin{aligned} \min \quad & \nabla f\left(\alpha^{(k+1)}\right)^T d \\ \text{sub. to} \quad & y^T d = 0, \\ & d_i \geq 0 \quad \text{for } i \text{ such that } \alpha_i^{(k+1)} = 0, \\ & d_i \leq 0 \quad \text{for } i \text{ such that } \alpha_i^{(k+1)} = C, \\ & -1 \leq d_i \leq 1, \\ & \#\{d_i \mid d_i \neq 0\} \leq n_c. \end{aligned}$$

Let $\bar{\mathcal{B}}$ be the set of these indices.

A4.2. Fill $\bar{\mathcal{B}}$ up to n_{sp} entries with indices $j \in \mathcal{B}$. Set $\mathcal{B} = \bar{\mathcal{B}}, k \leftarrow k+1$ and go to A2.

Figure 2. Parallel Gradient Projection Decomposition Technique Algorithm

III. SMC-PBC-SVM Results Analysis

The SMC-PBC-SVM algorithm was implemented using the object-oriented C++ programming language and the standard MPI communication routines [19]. The experiments are carried out on two different parallel platforms at the Mississippi Center for Supercomputing Research (MCSR) [20], and ALBACORE Linux clusters [21]. The performance analysis was visualized using Jumpshot software [22]. The best performance was achieved with ALBACORE, which contains 12 compute nodes, each node has 2 chips, and each chip has 4 cores. Each core is an Intel(R) Xeon(R) CPU X5570 with 2.93 GHZ and 8192 KB Cache. Each node of node0 and node1 has 16 GB, and node2 to node11 has 12 GB.

The SMC_PBC_SVM has been tested using different size datasets, Earthworm, Protein, Mnist, and Mnist8m, which are small, medium, large, and very large datasets respectively. Table I includes the description of these datasets. The results show that SMC-PBC-SVM is very efficient with very large datasets as Mnist8m dataset, highly efficient with large datasets as Mnist dataset, reasonably efficient in medium datasets as Protein dataset, and less efficient with small datasets as Earthworm dataset. For more details, see Table II, which shows the training run time when number of processes 1, 2, 4, 9, 16, 25, and 36 are used.

TABLE I. Datasets Description

Dataset Name	Earthworm	Protein	Mnist	Mnist8m
Reference	[23] LY10	[24] JW02	[25] YL98	[26] GL07
Classes Number	3	3	10	10
Features Number	869	357	780	784
Training Samples Number	248	17766	60000	8100000
Testing Samples Number	30	662	10000	10000
Best C	32.0	32.0	32.0	32.0
Best γ	0.001953	0.001953	0.001953	0.001953
Accuracy %	100	69.55	98.21	98.73

TABLE II. Training Run Time in Seconds

Processes Number	Earthworm	Protein	Mnist	Mnist8m
NP = 1	460.000	2427.840	4118.800	5111.370
NP = 2	454.031	2299.361	3915.277	4641.645
NP = 4	421.053	1618.583	2187.881	2961.146
NP = 9	398.174	933.000	1409.055	1824.251
NP = 16	367.177	732.000	986.758	1252.153
NP = 25	347.444	516.617	814.520	945.837
NP = 36	334.620	431.059	701.696	768.011

The complexity for the multi-class classification is $O(K\,M\,N^2)$, where K is the number of classes, M is the number of features, and N is the number of training samples [3]. This serial complexity is the worst case scenario for multiclass classification using binary classifiers. But when this job is distributed among P processors, the parallel complexity becomes $O\left(\frac{K\,M\,N^2}{P} + T_c\right)$, where T_c is the complexity due to communication for task scheduling and combining the results.

We evaluate the parallel performance by the relative speedup (S), which is defined as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with p identical processing elements, then

$$S = \frac{T_s}{T_p} = \frac{\text{Training Time spent on a single processor}}{\text{Training Time spent on a Np processors}} \quad (4)$$

Then,

$$S = O\left(\frac{K M N^2}{\frac{K M N^2}{P} + T_c}\right) \quad (5)$$

$S \cong O(P)$, if $T_c \to 0$,
when transmitted data between processors is insignificant.

Efficiency (E) is another way to analyze the parallel implementation, which is defined as the ratio of speedup to the number of processing elements, then

$$E = \frac{S}{P} = \frac{\text{Speedup}}{\text{Processes number}} \quad (6)$$

Then,

$$E = O\left(\frac{\frac{K M N^2}{\frac{K M N^2}{P} + T_c}}{P}\right) \quad (7)$$

$S \cong O(1)$, if $T_c \to 0$,
when transmitted data between processors is insignificant.

Figures 3, 4, and 5 show the relationship between the number of processors and run time, speedup, and efficiency respectively. Also, Figures 6 and 7 show the output of Jumpshot, which is a visualization tool to study the performance of parallel programs using log files that are generated from the execution of the SMC-PBC-SVM implementation to Earthworm and Mnist datasets using 16 processors.

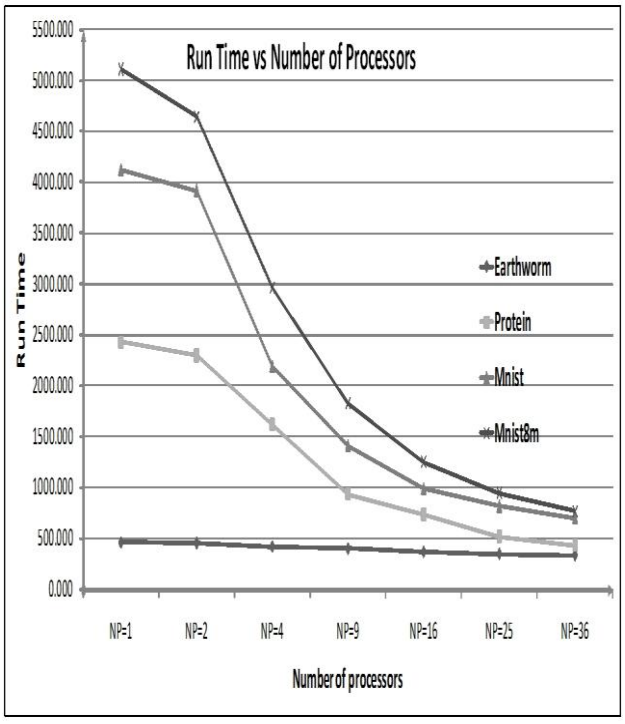


Figure 3. The relationship between Run Time and Number of Processors

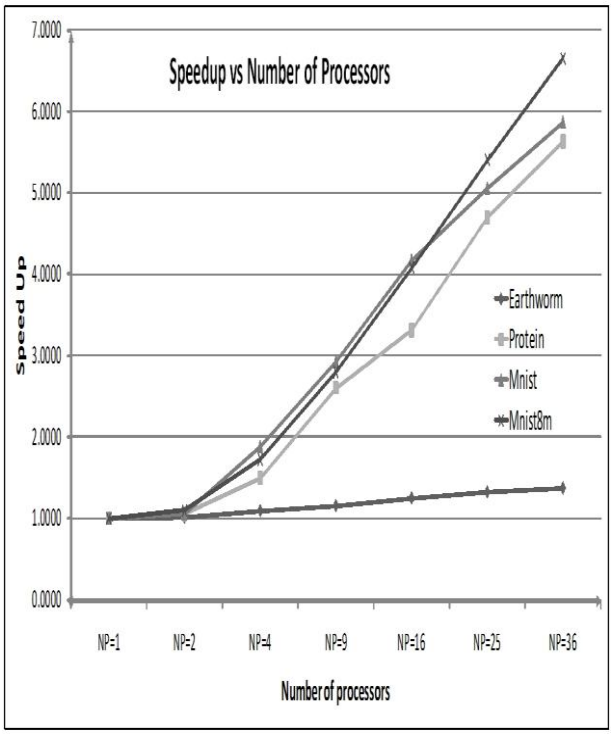


Figure 4. The relationship between Speedup and Number of Processors

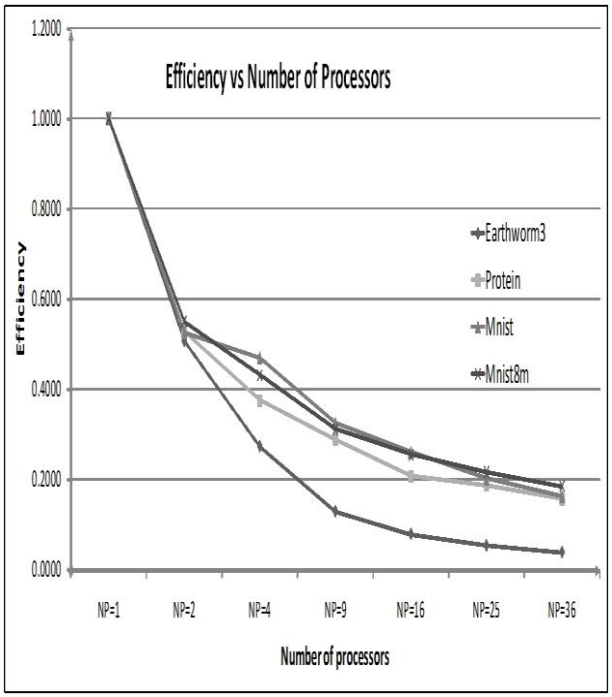


Figure 5. The relationship between Efficiency and Number of Processors

In Figure 3, we can see that the training execution time goes down as the number of processors is increased. It shows that when there is a sufficient work to be done concurrently on an increasing set of processors, there is a related improvement in performance. For small datasets, as Earthworm, there is not

enough work to be done concurrently. Therefore, there is no significant improvement in performance in that case. On the other hand, for very large datasets, as Minst8m, there is a sufficient work to be done concurrently. Therefore, there is a significant improvement in performance with very large datasets. By the same concept, the performances of medium and large datasets, such as Protein and Mnist, fell between the performances of small datasets and very large datasets. We used a Portable batch system (PBS) script which allows us to choose number of processors which are needed for job execution.

Referring to the speedup (5), we can see that, the size of datasets and the size of classes are significant factors in speedup. Therefore, it may not be possible to avoid the idling of some processes. Also, as the number of processes is increased, the speedup is less than linear, indicating idle time for few processors while waiting for other processors. In Figure 4, we can observe that the speedup is closed to linear speedup from very large dataset to small dataset.

Efficiency (7) is defined as the ratio of speedup to the number of processors. Therefore, a higher speedup ensures good efficiency, implying efficient use of the parallel resources. In Figure 5, we can see, the efficiency of SMC-PBC-SVM is gradual from a very large dataset to a small dataset.

In Figure 6, we can observe that there is a lot of lost time through idling especially with a large number of processes where the dataset size is small. Therefore, this explains the low efficiency with small datasets. While in Figure 7, we can see that there is no lost time because most time is spent in computation without idle time, where the dataset size is large. Therefore, this explains the high efficiency with large datasets.

IV. CONCLUSION AND FUTURE WORK

In this paper, the problem of solving multi-class classification using an efficient parallel support vector machine implementation was investigated. SMC-PBC-SVM is an efficient parallel algorithm, which combines Parallel Binary Class with Serial Multi-Class Support Vector Machines for classification. The SMC-PBC-SVM algorithm was implemented using the object-oriented C++ programming language and standard Message Passing Interface (MPI) communication routines. The parallel code was executed on an ALBACORE Linux cluster, and then tested with four datasets with difference sizes: Earthworm, Protein, Mnist, and Mnist8m. The results show that the SMC-PBC-SVM implementation can significantly improve the performance of data classification without loss of accuracy. As the dataset becomes larger, the SMC-PBC-SVM achieves a higher efficiency. In this paper, we used one processor in mulgroup which means SMC, and more than one processes in bingroup which means PBC.

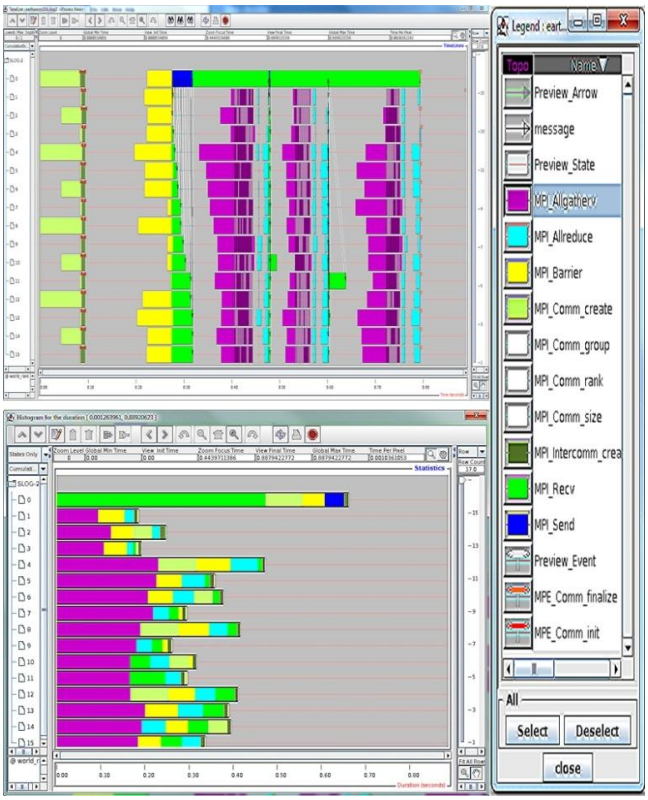


Figure 6. Jumpshot Timeline, Histogram and legend windows of Earthworm dataset using 16 processors

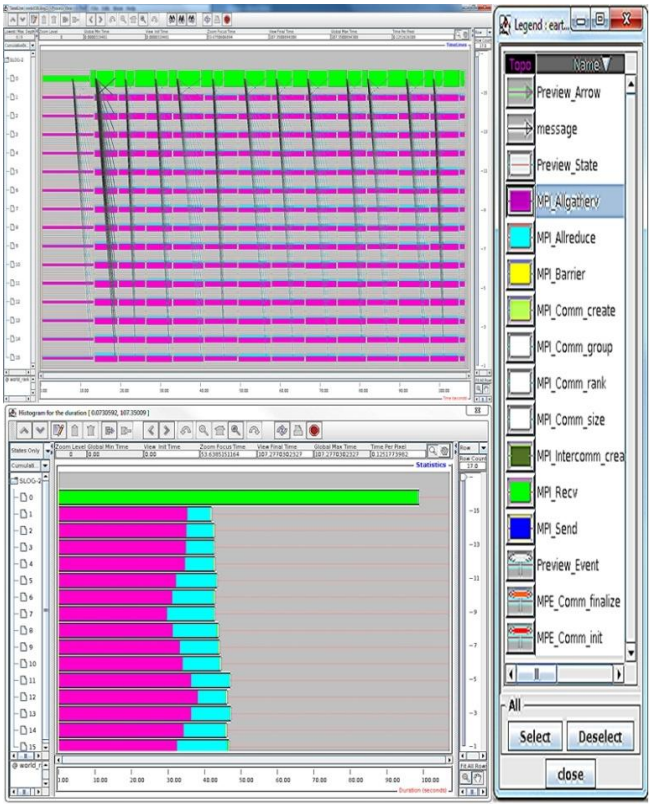


Figure 7. Jumpshot Timeline, Histogram and legend windows of Mnist dataset using 16 processors

In the future work, the scope of mulgroup will extend to include more than one processor, (Add comma) where each of which has its own bingroup. Therefore, these processes in mulgroup work in parallel, and then we will produce PMC-PBC-SVM implementation which will improve the performance and will address the limitations of SMC-PBC-SVM.

ACKNOWLEDGMENT

The authors gratefully acknowledge the Beni Suef University for their financial support.

REFERENCES

[1] C. Cortes, and V. Vapnik, "Support-vector networks Machine Learning," 1995.

[2] V. Vapnik, "The Nature of Statistical Learning Theory," Springer-Verlag, 1999.

[3] A. Rajendran, "Parallel Support Vector Machines for Mulicategory Classification of Large Scale Data," Dissertation, University of Southern Mississippi, 2007.

[4] E. Osuna, R. Freund, and F. Girosi, "An improved training algorithm for support vector machines," IEEE Workshop Neural Networks for Signal Processing, pp. 276–285, 1997.

[5] S. Qiu, and T. Lane, "Parallel computation of RBF kernels for support vector classifiers," Fifth SIAM International Conference on Data Mining, 2005.

[6] G. Zanghirati, and L. Zanni, "Parallel solver for large quadratic programs in training support vector machines," Parallel Computing 29, pp. 535-551, 2003.

[7] L. Zanni, T. Serafini, and G. Zanghirati, "Parallel software for training large scale support vector machines on multiprocessors systems," Journal of Machine Learning Research 7, pp. 1467-1492, 2006.

[8] T. Serafini, G. Zanghirati, and L. Zanni, "PGPDT: Parallel Gradient projection based on Decompostion Technique," Technical report, http://dm.unife.it/gpdt/,2007.

[9] C. Zhang, P. Li, A. Rajendran, and Y. Deng, "Parallelization of multicategory support vector machines for classifying microarray data," BMC Bioinformatics, 2006.

[10] T.Eitrich, and B. Lang, "Efficient Implementation of serial and parallel support vector machine training with a multi-parameter kernel for large-scale data mining," Proceedings of World Academy of Science, Engineering, and Technology 11, 2006.

[11] T. Serafini, L. Zanni, "On the Working Set Selection in Gradient Projection-based Decomposition Techniques for Support Vector Machines," Optim. Meth. Soft. 20, pp. 583-596, 2005.

[12] R. Fan, P. Chen, and C. Lin, "Working set selection using second order information for training support vector machines," Journal of Machine Learning Research, 2005.

[13] T.Eitrich, and B. Lange, "On the optimal working set size in serial and parallel support vector machine learning with the decomposition algorithm," Proceedings of the fifth Australasian Conference on Data mining and analysis 61, pp. 121-128,2006.

[14] J. Platt, "Fasting training of support vector machines using Sequential Minimal Optimization," In Advances in Kernel Methods Support Vector Learning, MIT press, pp. 185-208, 1999.

[15] J. Dong, A. Krzyzak, and C. Suen, "A fast parallel optimization for training support vector machine," Proceedings of 3rd Int. Conf. Machine Learning and Data Mining, pp. 96-105, Germany, 2003.

[16] J. Dong, A. Krzyzak, and C. Suen, "Fast SVM training algorithm with decomposition on very large data sets," IEEE Transactions on Pattern Analysis and Machine Intelligence 27, pp. 603-618, 2005.

[17] C. Change and C. Len, "LIBSVM: a library for support vector machines," Technical report, http://www.csie.ntu.edu.tw/_cjlin/libsvm/, 2005.

[18] T. Joachims, "SVM ^{Light}, Support vector machine," Developed at the university of Dortmund , and it is available at http://svmlight.joachims.org/, 1994.

[19] MPI: A Message Passing Interface Standard (Version 2.2), Message Passing Interface Forum, September 4, URL http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, 2009.

[20] MCSR: Mississippi center for supercomputing research, Technical report, http://www.mcsr.olemiss.edu.

[21] USM: Albacore Cluster, Chemistry lab at The University of Southern Miss. http://albacore.st.usm.edu/cgi-bin/portal.cgi.

[22] A. Chan, D. Ashton, R. Lusk, and W. Gropp, "Jumpshot-4 Users Guide," Mathematics and Computer Science Division, http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/index.htm, 2007.

[23] Y. Li, N. Wang, E. Perkins, C. Zhang, and P. Gong, "Identification and Optimization of Classifier Genes from Multi-Class Earthworm Microarray Dataset," PLoS ONE 5(10): e13715. doi:10.1371/journal.pone.0013715, October 2010.

[24] J. Wang, "Application of support vector machines in bioinformatics," Master's thesis, Department of Computer Science and Information Engineering, National Taiwan University, 2002.

[25] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proceedings of the IEEE, 86(11):2278-2324, MNIST database available at http://yann.lecun.com/exdb/mnist/, 1998.

[26] G. Loosli, S. Canu, and L. Bottou, "Training invariant support vector machines using selective sampling," In Léon Bottou, Olivier Chapelle, Dennis DeCoste, and Jason Weston, editors, Large Scale Kernel Machines, pages 301-320. MIT Press, Cambridge, 2007.

An Improved Design Technique for Cost Optimization in Distributed Database Systems (DDBSs)

Hassan I. Abdalla, Ali A. Amer
Department of Computer Science, College of Computer & Information Sciences, King Saud University, P.O. Box 51178, Riyadh 11543, Saudi Arabia
habdalla@ksu.edu.sa, aliaaa2004@yahoo.com

Abstract- *Horizontal and vertical partitioning are important aspects of physical design in relational database system that has a significant impact on performance and cost optimization. However, the distribution design also involves making decisions on the fragments allocation across the sites of a computer network. In this paper we address the allocation and re-allocation phases of distributed database systems by adopting an efficient algorithms that leads to a proper data allocation schemes. A new data allocation algorithm for DDBSs is presented in this work, the proposed Algorithm explores and improves some concepts used in previously developed algorithms to reallocates fragments to different sites by adopting the shortest path method to transfer fragments between the old location and the new one when migration decision is made. Experimental results proves that the proposed algorithm efficiently reduces transmission cost and speed of fragment migration over the network sites improving the overall DDBS performance.*

Keywords: Allocation, DDBMS, Partitioning, Distribution, Optimization.

1 Introduction

The allocation technique is the process of placing fragments to different sites after the database has been partitioned [13]. However, because this process plays a key role in DDBSs performance, it needs to be performed in an optimal way to reduce communication cost during query execution to achieve the proposed objectives.

There are several allocation techniques that have been employed in [2,7, 12,15,20] for dynamic allocation in distributed database systems to improve performance. They addressed dynamic environments, where sites access probabilities to fragments change over time. The objective of this paper is to design an effective allocation algorithm that guarantees minimum data transfer cost in a dynamic environment.

The allocation problem of the DDBS had been studied extensively in [6, 11] for redundant and non-redundant scenarios. In [5,2], data allocation is performed in static environment where site access probabilities to fragments never change. However, this static allocation would degrade the database performance if these access probabilities change.

A framework for dynamic data allocation was presented in [3]. A query processing cost model to evaluate the performance of the system is covered in the context of complex value databases in [10] in which a heuristic approach was proposed for performing fragmentation and allocation. In [7] a model for dynamic data allocation and redistribution was given. [4] suggested incremental allocation and reallocation based on changes in workload. SAGA approach was proposed in [1] to determine the optimal places for data fragments where data access pattern and transmission cost were used to help in specifying the optimal places.

A dynamic object allocation and replication algorithm with centralized control has been proposed in [12]. In [16] an optimal algorithm for non-replicated database systems was presented. In [15, 20] two algorithms for non-replicated distributed databases were proposed, while [20] used a threshold algorithm where the fragments are continuously reallocated according to changes in data access patterns while [15] added time constraint to reallocates data with respect to the changing data access patterns. A more comprehensive algorithm for a dynamic fragment allocation that incorporates time constraints, threshold value and the volume of data transmitted

to reallocate fragments to sites dynamically at runtime was presented in [12].

In [8] data allocation model was presented to maintain and solve DDBS problems by studying the initial allocation and post allocation problems. [9] Presented a solution for data allocation and optimization in distributed database systems that performed by means of mobile agents. A decentralized approach for dynamic table fragmentation and allocation in DDBS based on observation of the sites access patterns had been proposed in [14] that performs fragmentation, replication, and re-allocation based on recent access history. An integer programming formulation for the non-redundant version of the fragment allocation problem is presented in [13].

Our proposed algorithm is considered as an optimization to the optimal algorithm of [2], a threshold algorithm of [19, 20], TTCA algorithm of [15] and synthesis algorithm of [12]. Thus, the proposed algorithm incorporates and improves the concepts proposed in [2,12,15,19,20]. Our proposed algorithm migrates fragment (F_i) to site (S_j) only if the following conditions are satisfied: (i) Site S_j has made highest query costs for fragment F_i than all other sites, (II) The average transfer cost of fragment F_i is greater than the threshold value, and (iii) The remote access counter for fragment F_i is greater than its local access counter over several time intervals. Moreover, the proposed algorithm adopts the shortest path technique when moving fragment F_i from the old location to the new location.

1.1 The Paper Contributions

This paper contribution can be summarized in the following; first, it re-allocates data fragments according to the fragment accesses given that sites constraints are not violated. Second, it calculates and chooses the threshold value based on data transmission costs (which adopted different calculation method compared to that used in previous algorithms). Third, calculating and using the query cost QC_i^j between sites S_i and S_j, and QC_i^k between site S_i and all other sites (excluding S_i) as an alternative to the data transmission volume method used in the synthesis algorithm. Forth, it adopts the shortest path algorithm when moving fragments between the old and the new locations to minimize the transmission cost.

1.2 Paper Organization

The rest of this paper is organized as follows; Section 2 presents the proposed methodology for non-replicated dynamic data allocation. The proposed algorithm is presented in section 3. A comparison based on performance factors is given in section 4. Finally, conclusions are given in section 5.

2 The Proposed Methodology

In our proposed algorithm all the advantages of the dynamic algorithms presented in [2,19,15,20,12] will be incorporated to produce a more efficient algorithm that minimizes transmission cost given the site constraints. It is believed that this technique will prove to be a potential progress not only in reducing transfer cost but also in enhancing the overall performance of the DDBSs.

2.1 DDBS Environment

Assuming that, we have a fully connected network that consists of M sites, where each site has N fragments that initially distributed in a static way as shown in Figure1. Every fragment F_i at site S_j has two variables, the LAC_i (representing the number of local accesses to fragment F_i at site S_j) and the RAC_i (representing the number of remote accesses to fragment F_i at site S_j). And every site S_j has two constraints: Capacity C_j (indicating that no site will receive more than its capacity) and Fragment Limit FL_j (representing the maximum number of fragments each site can handle). The following migration conditions need to be satisfied for fragment F_i to be moved from S_h to site S_j.

$$S_j.F_i.RAC > S_j.F_i.LAC, \quad j = 1,2,\ldots, m \qquad (1)$$

$$\sum_{j=1}^{m} QC_h^{\,j} > \sum_{i=1}^{n} QC_h^{\,k}, \quad k,h = 1,2,\ldots, m, m <> j \qquad (2)$$

$$\sum_{j=1}^{m} QChj = DRhj * TChj \quad (3)$$

$$FTC = \left(\sum\left(\mathrm{Req_{ij}} * \mathrm{TC_{hj,}}\right),\ 1 <= \mathrm{j},\ \mathrm{h} < \mathrm{m}\right) \quad (4)$$

$$Threshold\ value = \left(\sum_{j=1}^{m}\sum_{i=1}^{n} Req_{ij} * TC_{hj} \right) / n \quad (5)$$

$$FTC > threshold\ value \quad (6)$$

Equation (1) states that the remote access counter should be greater than local access counter for fragment *(F_i)* under consideration. Equation (2) states that the average query cost between site S_h *and* site S_j should be higher than average query cost between site S_h and all other sites accessing fragment F_i (excluding site S_j). Equation (3) is used to compute the query cost between site S_h and site S_j (the same equation can be used to compute the total query cost between site S_h and all other sites). the query cost between sites S_h and S_j can be calculated by multiplying the transmission cost unit between them (TC_{hj}) by the resulted volume of data (DR_{hj}) which represents the data obtained by executing site S_j queries that require a particular data volume from fragment F_i allocated at site S_h). Equation (4) is used to give fragment transmission cost across sites S_h and S_j for fragment F_i. Equation (5) calculates the threshold value for fragment F_i. Equation (6) states that the fragment transmission cost should be greater than the threshold value.

The following constraints have to be considered throughout the allocation process.

$$\sum_{i=1}^{n} q_{ij} * Z_j < C, \qquad 1 \le j \le m \quad (7)$$

$$\sum_{i=1}^{n} q_{ij} = 1,\ 1 \le i \le n \quad (8)$$

$$\sum_{j=1}^{m} q_{ij} < FL, 1 \le i \le n \quad (9)$$

Equation (7) represents the capacity constraint which indicates that no site will receive more than its capacity. Equation (8) states that a fragment will be allocated to only one site. Equation (9) states that each site will not handle more than a given number of fragments denoted by fragment limit (*FL*). Table 1, shows the notation used in this work.

TBALE1: The Notation Used

RAC	Remote Access Counter
LAC	Local Access Counter
F	The fragments DDBSs.
S	DDBSs sites
C	Site capacity
Z	The data fragment size in DDBSs
FL	Fragment limit for each site
FI	The accessed Fragment Identifier
Q	Site query
I	Fragment index
N	Fragments number
M	Sites number
K	Query index
J	Site index
ASA	Accessing Site Address
CSA	Candidate Site Address
AC	Access Counter for each accessed site
TFA	Time of the fragment accessed
TCS	Time the candidate site address has been selected
DR_{ij}	Fragment size (in bytes) resulting from site S_i query accessing fragment F_i located at site S_j
AT	Access Time for each accessed site
QC_h^j	Average of query cost between S_h and S_j
QC_h^k	Average of query cost between S_h and any other site S_k
Req_{ij}	Equal 1, if F_i is required by S_j and 0, otherwise
RF_{ri}	Retrieval frequency of retrieval operation from site j
UF_{uj}	Update frequency of update operation from site j
QF_{ij}	Access frequency of the i^{th} query at site j
V	Volume of fragment allocation i (characters)
SC	Storage cost ($ / 5,OOO char/month)
X_{ij}	Equal 1, if F_i allocated to site S_j and 0, otherwise
TC_{ij}	Cost for site *Si* accessing a fragment located at site *Sj*

3 The Proposed Algorithm

The proposed algorithm is performed in the following three steps:

A. Determining the shortest path and their values.
B. Initializing and determining variables.
C. Running the proposed algorithm while regularly checking sites constraints.

3.1 Determining the Shortest Paths

Occasionally, after any network topology change the algorithm begins by running *Dijkstra's* algorithm of [10] in every site to obtain the shortest path from one site to every other site. This algorithm is consistently repeated until the shortest path between sites is obtained. Having a weighted directed graph *G = (V, E)*, with a weighted function *W: E → R* mapping edges of real valued weights with sites S_1, S_2, S_3 and S_4 (see Figure 1). The

shortest path matrix would look like the one in table 2.

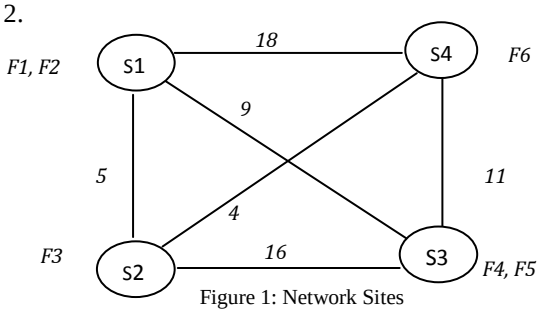


Figure 1: Network Sites

TABLE 2: Shortest Path Matrix

Source Site	Destination Site	Path	Path value
S1	S2	S1----S2	5
S1	S3	S1----S3	9
S1	S4	S1---S2---S4	9
S2	S3	S2---S1---S3	14
S2	S4	S2----S4	4
S3	S4	S3----S4	11

3.2 Threshold Calculation

Each fragment is accessed by at least one site. Accesses are represented by the Access Cost Matrix (*ACM)*. $ACM_{i,j}$ gives the number of times site S_j accesses fragment F_i. In this work, based on the Site Access Record (*SAR) shown in the paper example (table 7),* the *ACM* matrix is constructed as shown in table 3.

TABLE 3: Fragments Access Cost Matrix (ACM)

S/F	F_1	F_2	F_3	F_4	F_5	F_6
S_1	1	1	0	1	0	0
S_2	1	3	1	0	2	2
S_3	2	0	0	1	1	1
S_4	0	1	2	0	0	2

The transmission cost matrix *TCM (table 4)* gives the cost of accessing fragment F_i by site S_j. Using these matrices, the threshold value is computed based on Fragment Usage Matrix (*FUM*) which is produced by multiplying *ACM* matrix by the *TCM* matrix. Using *FUM*, we select the site whose average access cost for fragment F_i is the highest among all other sites for the same fragment. An illustrating example is given next.

TABLE 4: The Transmission Cost Matrix (TCM);

Sites	S1	S2	S3	S4
S1	0	5	9	18
S2	5	0	16	4
S3	9	16	0	11
S4	18	4	11	0

3.3 Initializing Variables

a. There are a number of *N* fragments distributed across *M* sites, where each site contains one or more fragments. Queries may access several fragments allocated at different sites. And each site has two constraints: fragment limit (*FL*) and site capacity (*C*) as shown in table5.

b. A separate data structure called Site Access Record (*SAR*) is kept for each site. The *SAR* stores information about fragments accesses at each site, denoted by SAR^k_j , indicating the. k^{th} access at site S_j, where (k = 1,2,3,…to unlimited access number, and j = 1,2,3,...m). *SAR* stores the following information for each access:

1. Accessed Fragment Identifier (*AFI)*, see *table 6*.
2. Accessing Site Address (ASA).
3. Data Record (DR) on executing a query *Q* from site S_i on fragment F_i located at site S_j (in bytes).
4. Access Time (AT) by site S_j to fragment F_i.
5. Access Counter (AC) to keep the number of access times for each accessed site.

c. Also for each site a data structure named Access Counter Record (*ACR*) is kept for every fragment in the site. The *ACR* record stores the following information about the fragment after each access:

i. Candidate Site Address (CSA): The address of the site that incurred a query cost value that is higher than that of all other sites over a time interval (*t1* to *t2*). Initially, *CSA* is set to the address of the site where the fragment is currently located.
ii. The number of local accesses to the stored fragment (*LAC)*.
iii. The number of remote accesses to the stored fragment (*RAC*).
iv. The time of the candidate site address selection (*TCS*).

For each locally stored fragment, initialize both the local and remote counters to zero (*LAC* = 0, *RAC* = 0).

3.4 Our Algorithm

After processing different site accesses to fragment F_i allocated at site S_j over time t, the following steps are performed:

```
Main {
// Enter SAR and ACR data
Enter-data ( );
// Calculate query costs
Compute- QC ( );
// Calculate threshold values
Compute-threshold (T1,T2,OCS: J, NCS: h )
// Check whether the accesses are local or remote
Checki-ASR ( );
// Make sure if the migration conditions satisfied
Migrate-data( );
}// end main

Enter-data ( ){
{for j=1 to m do
  For h = 1 to k do // Enter site data
   Read(S[j].AC,  S[j].FI,  S[j].ASA,  S[j].AT,
S[j].DTV)
End h
For i=1 to n do //Enter fragment data
  Read(S[j].F[i].CSA, S[j].F[i].DCA)
End i
End j}

Compute- QC (T1,T2, old site: J, new site h ){
{Sj=0; Sall=0; // the sum performed among T1 andT2
For l=1 to m do
For i=1 to k do
If ((S[j].ASA= h) and (AT between (T1,T2)))
Sj=Sj+ S[L].DRjh* TCjh// QC for Si and Sh

Else //  If ((S[l].ASA<> h) and (AT between
(T1,T2)))
Sl= Sl+ s[j].DRjl * TCjl ; //QC for Si and all others

End for i
End for l
// test QC condition
If (Sj>Sl) return true
Else return false}
End for l

Compute- threshold ( ){
{// Build FUM matrix using ACC and TC
// Enter number of fragment Fi and the targeted site
for which threshold value will be calculated
Thv  =  FUM(i,  h);  S=0;  //threshold  value
assignment
For j = 1 to m do
// cumulative access cost for all sites other than Sh
If j < > h then
S = S + FUM (i, j)
End i
If ( S/(m-1) > thv) //threshold condition satisfied
Return (false)
Else return (true)
End j}

Checking-ASR ( ){
{// determine local access from remote access
if S[j].ASA='k'
  {If S[j].F[i].CSA<>'k'
     F[i].CSA='k';
     LAC++; // local access}
Else
 {{if S[j].F[i].CSA<>'k' then
    F[i].CSA='k';
    RAC++;   // remote access }}

SPA(CCM);{
// output is the shortest path matrix
For k=1 to m do
 For i=1  to m do
  For j=1  to m do
   valueij = minimum (ccij,ccik + cckj)
   keep (SPA[i],SPA[j], valueij);
 End for j
End for i
End for k}

Migrate-data{// fragment migration decision
If ((F[i].RAC > F[i].LAC) && Compute- QC &&
Compute- threshold) then
If (there is no site constraints violation) then
 SPA(CCM); // to produce the shortest path matrix
 Migrate (F[i],OCS,NCS, SPA(OCS,NCS));
Else
{Reset-zero(F[i].RAC, F[i].LAC);
Cancel fragment migration;}
}// end migrate-data
```

4 Comparing Algorithms Based On Performance Factors

The comparison of our algorithm with the existing (Optimal, Threshold, TTCA and Synthesis) algorithms presented in [2], [19], [15,20] and [12] respectively, is clearly showed in (*Figure 3*) based on the following factors:

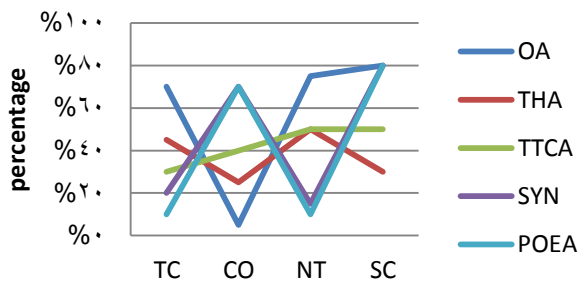
1. Fragment migration decision
2. Storage cost (*SC*).
3. Transmission cost (*TC*), Computation overhead (*CO*) and Network Traffic overhead (*NT*).

1. **Fragment migration decision**: the optimal algorithm of [2] moves fragment F_i when the counter for the remote site in the access matrix is greater than the counter of the owning site for that fragment. The threshold algorithm migrates the fragment when the counter for fragment F_i is greater than the threshold value. The TTCA algorithm migrates fragment F_i when the counter of the remote site is greater than the threshold value (t) and all the last "t+1" accesses made in a specified time (T). In synthesis algorithm of [12], the fragment is migrated to the intended site when that site has made data transfer cost that is higher than all other sites and is greater than the threshold value consistently in some recent time intervals. However, our algorithm migrates the fragment when the remote counter for fragment F_i is greater than the local counter of the owning site and the resulted query cost (*QC)* between the old site and new site is greater than the query cost between the old site and any other accessing site for that intended fragment, given that the shortest path between the old site and the new site is maintained.

2. **Storage cost factor**: it is clear from Figure 3, that the optimal algorithm of [2] consumes much storage space since it stores several access counters for each fragment. The threshold algorithm requires less storage space as compared to the optimal algorithm because it stores only one counter for each fragment. The TTCA algorithm requires more storage space as compared to optimal and threshold algorithms. The synthesis algorithm of [12] needs high storage space and thus cost because it maintains several data structures. Our algorithm requires more storage space compared to the algorithms of [2, 19, 20, 15] and almost the same space as that of [12]. This high space requirement is again because our algorithm maintains several data structures.

3. **Transmission cost and computation overhead**: Figure 3, clearly shows that the optimal algorithm sustains the highest transmission cost. This is because the basic rule of thumb that, the more access frequency acquired, the more transmission cost and network traffic overhead will be incurred. The threshold algorithm minimized the network traffic and transmission cost as a result of using the threshold factor. It also obtained less computation overhead compared to optimal algorithm. However, the TTCA algorithm used time constraint and threshold value to further decrease the network traffic and transmission cost compared to optimal and threshold algorithms. More computational overhead is produced for the synthesis algorithm of [12], due to the total volume computation for a fragment transmission to all sites accessing it within a time interval (τ). Our proposed algorithm minimizes the transfer cost for fragments migration dramatically compared to other algorithms because of the following added features: the remote counter (has to be greater than local counter for the transmitted fragment F_i). The threshold value (differently computed compared to that used in the other algorithms). Query cost (*QC*) calculation (the query cost QC_h^j between sites S_h *and* S_j and QC_h^k between site S_h and all other sites excluding site S_j) compared to data transmission volume used in synthesis algorithm, and finally the adoption of the shortest path method. According to our proposed algorithm, the fragment will stay at the owning site as long as there are site constraints violations.

Figure (2): DDBS Performance factors



4.1 A Practical Example

The following example is to test the validity of our algorithm. In this example there is a network of four sites in which six fragments are initially distributed according to a random allocation method. Our proposed re-allocation method will be tested based on the information presented in *tables (5 and 6)*.

TABLE 5: DDBS Fragments

Fragment	F_1	F_2	F_3	F_4	F_5	F_6
Identifier	1	2	3	4	5	6
Size	810B	620B	900B	660B	521B	701B

TABLE 6: DDBS Sites

Site	S_1	S_2	S_3	S_4
Capacity	2350B	1650B	2020B	3200B
Fragments Limit	3	1	3	4

First; we apply the minimum algorithm of [18] on the communication cost matrix to obtain the shortest path matrix. Then we keep the shortest paths values based on the given network graph as shown in section **III.A.**

Second; assuming that multiple accesses are performed, the site access record (*SAR*) is constructed as shown in table 7.

TABLE 7: Sites Access Records SAR_i^j

AC	SN	FI	ASA	AT	DR
1	S_1	1	1	2:10	30B
2	S_1	1	2	3:05	28B
3	S_1	1	3	3:10	27B
4	S_1	1	3	2:00	31B
5	S_1	2	1	3:10	33B
6	S_1	2	2	3:10	33B
7	S_1	2	2	3:12	12B
8	S_1	2	2	2:10	20B
9	S_1	2	4	3:00	26B
1	S_2	3	4	2:10	25B
2	S_2	3	2	3:00	25B
3	S_2	3	2	4:05	41B
1	S_3	4	1	1:50	43B
2	S_3	4	3	2:10	39B
3	S_3	5	3	3:00	28B
4	S_3	5	2	4:05	31B
5	S_3	5	2	4:50	29B
1	S_4	6	2	3:10	32B
2	S_4	6	2	3:50	34B
3	S_4	6	3	3:10	28B
4	S_4	6	4	3:10	20B
5	S_4	6	4	3:12	20B

Based on the sites access records (*SAR*), the Access Cost Matrix (*ACM*) will be constructed as mentioned earlier and shown in table 3 above.

Third: The *ACM* matrix is multiplied by *TCM* matrix to compute fragment usage matrix (*FUM = ACM * TCM*), *FUM* matrix is presented in *table 8*;

TABLE 8: Fragments Usage Matrix (FUM)

F/S	S_1	S_2	S_3	S_4
F_1	23	37	25	44
F_2	33	9	54	58
F_3	41	8	38	4
F_4	9	21	9	29
F_5	19	16	32	19
F_6	55	24	54	19

Based on *FUM* matrix, the threshold value can be calculated for fragments and sites individually when needed (*table 9*). Finally, based on threshold values and the access counter records ACR, the migration decision will be made for each fragment as shown in *table 9*.

TABLE 10: Migration Decision

F_I	Migration (Y/N)	Constraints violations (Y/N)	Threshold value	Migration Fails (F)/successes (S)
1	N	N	24, 34.6	F
2	Y	N	45,5, 32	S
3	N	N	-	F
4	N	N	-	F
5	Y	Y	24, 23.3333	Migration fails because violation of S_2 constraints
6	Y	N	36,75, 32.68	S

5 Conclusions

In this paper, we have presented a novel approach that handles data allocation problem during the design of distributed databases. A comparative study for different data allocation algorithms has been conducted to reach to the most efficient allocation scheme that leads to a better performance and less communication cost.

Our proposed algorithm is the most efficient one among all compared algorithms for dynamic data allocation as it has improved the *DDBS* performance by maintaining strict restrictions for fragment re-

allocation causing a significant reduction in the frequency of fragment migrations across sites which minimized network traffic and reduced data transmission cost by adopting the shortest path algorithm for data movement. However, the only drawback of our algorithm is that it requires more storage space relevant to some of the compared algorithms. However, this is considered as a very trivial drawback due to the dramatic fall down in the storage hardware prices.

Future work is in progress towards extending this algorithm to optimize non-replicated inter-active distributed database systems.

6 Acknowledgment

The authors would like to thank and appreciate the support received from the Research Center of the College of Computer & Information Sciences at King Saud University for providing the necessary facilities to accomplish this work.

7 References

[1] H. Abdalla, "An Efficient Approach for Data Placement in Distributed Systems", 2011 Fifth FTRA International Conference on Multimedia and Ubiquitous Engineering.

[2] Brunstrom, A., S.T. Leutenegger and R. Simha, 1995. Experimental Evaluation of Dynamic Data Allocation Strategies in a Distributed Database With Changing Workloads, in Proceedings of the 1995 International Conference on Information and Knowledge Management, Baltimore, MD, USA.

[3] Wilson, B. and S.B. Navathe, 1986. An Analytical Framework for the Redesign of Distributed Databases, in Proceedings of the 6th Advanced Database Symposium, Tokyo, Japan.

[4] A. Chin, "Incremental Data Allocation and Reallocation in Distributed Database Systems," Journal of Database Management, Vol. 12, 01.

[5] Cheng, C.H., W.K. Lee and K.F. Wong, 2002. A Genetic Algorithm-Based Clustering Approach for Database Partitioning, IEEE Transactions on Systems Man and Cybernetics, 32: 215-230.

[6] Chang, C.T., 2002. Optimization Approach for Data Allocation in Multidisk Database, European J.Operational Res., 143: 210-217

[7] S. Ceri and G. Pelagatti, "DDB Principles & Systems", McGraw-Hill International Editions.

[8] NH, Daudpota, Five steps to construct a model of data allocation for distributed database systems.Journal of Intelligent Information Systems: Integrating Artificial Intelligence and Database Technologies. 1998 vol.11, no.2.

[9] H. Grebla, G. Moldovan, S. A. Darabant, A. Câmpan, Data Allocation In Distributed Database Systems Performed By Mobile Intelligent Agents, Proceedings of the International Conference on Theory and Applications of Mathematics and Informatics - ICTAMI 2004, Thessaloniki, Greece.

[10] T. H. Cormen, C. E. Leisorson, R. L. Rivest, C. Stein, Introd to Algorithms 2nd Ed, Mc Hill 01.

[11] Eswaran, K.P., 1974. Placement of Records in a File and File Allocation in a Computer Network, in Proceedings of IFIP Congress on Information Processing, Stockholm, Sweden, pp: 304-307.

[12] Nilarun Mukherjee ,"Synthesis of Non-Replicated Dynamic Fragment Allocation Algorithm in Distributed Database Systems", Proc. of Int. Conf. on Advances in Computer Science 10.

[13] S. Menon, Allocating Fragments in Distributed Databases, IEEE transactions on parallel & distributed systems, vol. 16, NO. 7, July 2005.

[14] J. O. Hauglid , N. H. Ryeng, DYFRAM: dynamic fragmentation and replica management in distributed database systems, Distrib Parallel Databases (2010) 28: 157–185 ,8 September 2010.

[15] Arjan Singh and K.S. Kahlon, "Non-replicated Dynamic Data Allocation in Distributed Database Systems", IJCSNS International Journal of Computer Science and Network Security, VOL.9 No.9, pp. 176-180, September 2009.

[16] L.S. John, "A Generic Algorithm for Fragment Allocation in DDBS", ACM, 1994.

[17] M. T. Ozsu and P. Valduriez, Principles of Distributed Database Systems, 2nd ed., Pr. Hall, 99.

[18] A. Tamhankar and S. Ram, "Database Fragmentation and Allocation: An Integrated Methodology and Case Study," IEEE Trans. Systems, Man and Cybernetics, vol. 28, no. 3, 98.

[19] T. Ulus and M. Uysal, "Heuristic Approach to Dynamic Data Allocation in DDBSs", Pakistan Journal of Information and Technology, 2(3): 2003.

[20] T. Ulus and M. Uysal, "A Threshold Based Dynamic Data Allocation Algorithm - A Markove Chain Model Approach", Journal of Applied Science, Vol. 7(2), pp 165-174, 2007.

Design of Content-based Forwarding over Large-scale Storage Network

Zhaomeng Zhu, Gongxuan Zhang, Yongping Zhang, and Jian Guo
School of Computer Science and Technology, Nanjing University of Science and Technology, Nanjing, Jiangsu, China

Abstract - *This paper describes a design of a content-based publish/subscribe mechanism attached to large-scale storage network. The network is built as an infrastructure to support wide-area sensors network, or the Internet of Things. We want to collect the scattered computing resources of each node in a massive storage network to take up the heavy task of dissemination of a great amount of data produced by variety of sensors. Be inspired by the virtualization technology, we separate task to different functional units called bricks which can be self-tuning in large-scale network that composed by inexpensive commercial hardware. And we use Bloom filter to record algorithm status for effective forwarding. With less cost we could add more flexible data distribution mechanism to massive storage network.*

Keywords: content-based forwarding; distributed; Bloom filter; publish/subscribe

1 Introduction

During last few years, content-based publish/subscribe system has becoming more and more popular [1]. In publish/subscribe model, those who want data (be called as messages or events) from some kind of sources do not receive all the data that being produced, but only a subset for what they are interested in. Data's producers are called publisher and those who have provided sets of conditions to indicate their interests are called subscribers. The process of selecting messages for reception and processing is called filtering. In a content-based publish/subscribe system, data are only delivered to a subscriber if the attributes of those data exactly match constraints pre-registered in the system by the subscriber.

Our original propose of introducing content-based publish/subscribe mechanism to a large-scale storage network is to bring storage system the ability of supporting the wide-area sensors network [2], or the Internet of Things [3]. With a massive storage system in the cloud, data produced by wide variety of sensors can be aggregated and disseminated. The basic way of getting aggregated data is the searching service provided by underlying storage: users asking for data using a series of conditions and receiving all those data that exactly meet the conditions. The shortcoming is obvious that users must make queries by themselves. For a large scale of sensors network, there may be some sceneries like, (1) a user need to listen to some kind of sensor's data, whose time of occurrence can not be predicted, (2) some urgent data need to be delivered to users as soon as possible and can not wait for querying, and (3) continuous data should be automatically and continuously transferred to users without frequently querying. It is necessary to add a content-based publish/subscribe service to such a massive storage system for such sceneries. With this service, the only thing a user need is to register a subscription in system, and the data meeting conditions will be pushed to specified address automatically when occur.

Many distributed schemes have been proposed for content-based forwarding in wide-area sensors network [4-6], but our design focuses on some different goals. We are trying to integrate the content-based network with the storage networks. According to our observations, the massive storage network always contains a mass of computing nodes whose scale varies form thousands to millions. We try to collect these scattered computing resources so that we can design an effective content-based forwarding mechanism for massive storage system to support the sensors network.

For such a systems, there are still some constrains special. First, we should make full use of each node's limited computing resources, but do not affect the normal procedures of storage. The massive storage is designed to built using inexpensive commercial hardware whose computing power is not strong, so the additional forwarding mechanism need to undertake the enormous amount of data's forwarding tasks but only use as few resources as possible on each single node. Second, since the network's stability is built form a large of unstable components, and taking into account the natural dynamic of P2P architecture, the design must have a strong fault-tolerant ability. That's is, for example, when some of the participating peers left or crashed, the forwarding procedure should not be affected. At last, because of (1) the massive-storage will be distributed in different geographical regions as a city-wide or even world-wide infrastructure to support the wide-area sensors network, (2) the traffic caused by procedures and the contents of data may vary over time, and (3) not all data produced by sensors need to be delivered immediately using content-based information, content-based forwarding mechanism should have the capacity of self-tuning to meet such demands. In this paper, we introduce a architectural design of building a robust content-based forwarding mechanism above the large-scale storage network.

2 Background

2.1 P2P Massive Storage

More and more massive systems are built using peer-to-peer technology now [7-11]. Each node has the same functionality in a P2P storage network. The benefits of using P2P architecture includes, (1) system has a naturally highly scalability and there is no single node will became the bottleneck of performance since there is no any master node, (2) deleting and adding node is simpler since all node is same in functionality, and (3) few nodes' failure can not endanger the whole storage system. With these advantages of a P2P massive storage, we build our storage system which designed to support wide-area sensors network on a P2P overlay network referenced to existing systems like OceanStore [11] and BitVault [8].

For now, the main problem of a P2P storage network is the conflict between the reliability and persistence needed by storage applications and the dynamics of a P2P network [12]. But we assume that our storage network is deployed in a controlled and relatively fixed environment since we will build such a storage system as a public infrastructure, therefore, by using p2p technology we can make full use of the advantages while eliminating its shortcomings.

2.2 Bloom Filter

A Bloom filter [13] is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. An empty Bloom filter is a bit array of m bits, all set to 0. There must also be k different hash functions defined, each of which maps or hashes some set element to one of the m array positions with a uniform random distribution. To add an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1. And to query for an element (test whether it is in the set), feed it to each of the k hash functions to get k array positions. If any of the bits at these positions are 0, the element is definitely not in the set – if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive (but with a very small probability). While risking false positives, Bloom filters have a strong space advantage over other data structures for representing sets. On the other hand, an interesting property for Bloom filter is that union and intersection of Bloom filters with the same size and set of hash functions can be implemented with bitwise OR and AND operations, respectively.

3 Preliminary

3.1 Design Goals

Though the forwarding algorithm should be designed as quickly as possible, the performance of the massive storage network is more related to the storage procedures. For example, if one node can process only a few hundred objects pre second, an advanced content-based forwarding algorithm which is capable of processing thousands of objects but need more resources including CPU clocks and memory spaces maybe not significant better than a slower one. So what we mostly focus is not the speed or the total time used to finish all steps of algorithm, but the usability of the storage system, i.e. the attached content-based forwarding services should not affect the other aspects of a good storage system. At the same time, we think separating task to several sub-tasks and distributed them to different nodes is better than doing all by one node. Because by this way we can drop the average CPU load per node, that is, building a massive storage network using less total cost and lower power consumption.

Resisting to the instability of system is another thing we must pay much attention to. Our storage network has been designed to build from a mass of inexpensive commodity components, such as normal PC, or even embedded system attaching large storage devices that need lower power consumption to work. The failures of component will be inevitably frequent. That is said, we cannot presume that each node involved in always works just as we want. Our design must include some mechanism to ensure the algorithm's enforcement.

3.2 Terminologies

An *object* contains the data producer want to store and some information to describe data. For instance, a camera may take photos and send them to the storage. The photos will be preprocessed to attach some information to indicate the time and location. Such kind of descriptive information is called *metadata*, and is usually arranged in a form of *key-value pairs*. We call keys in key-value pairs the object's *attributes*.

Subscriptions are the information registered in the storage system by users who want data. A subscription contains a *disjunction* of *conjunctions* of *predicates* [14] and a forwarding *address*. A conjunction is also called a *filter*. For instance, subscription {{"Location" = "A5" ∧ "Time" > 20 ∨ "Type" = "image" ∧ "Producer" IN {"Camera-30A4", "Camera-30A5"}}, address = user0@example.com} has two filters, {"Location" = "A5" ∧ "Time" > 20} and {"Type" = "image" ∧ "Producer" in {"Camera-30A4", "Camera-30A5"}}, and every object that make all filters true will be sent to user0@example.com.

Predicates are pairs of *constrains* and its related attribute's name. For example, ["Location" = "A5"] is a predicate with a constrain of ["=", "A5"] and an attribute with name "Location". *Constrains* are atom units for checking and each contains a value and an *operation*. Different types have different operations. Every operation need two operands and produces a Boolean value. We call one value *satisfy* a one constrain if the result is *True* by feeding the attribute's value (in left) and the value in constrains (in right) to its operation.

For purpose of this paper, we consider two common types of data: *number* and *string*. We do not discuss *Boolean* since it can be represented by integer. We assume attribute name is a string. In this paper, we consider '<', '=' and '>' for number and on string we defined '=', '*prefix of*' (shorten it to '*PF*'), and "*IN*". An "IN" operation finds out if one string is in a set of strings.

Update the Bloom filter "a" with "b" is to add items in b into a since a bloom filter represents a set. This can be done by a bitwise OR operation.

4 Design

4.1 Overview

When an object arrives, the metadata will be picked out and assign an id. An id is a unique key generated by storage to identify an object. We only use this information but no actual data to reduce network traffic. After whole algorithm, we will get a set of forwarding addresses and send object's id to those addresses. Users can then use id to fetch data. By this way the storing procedure and the forwarding procedure can be handled separately and executed in parallel.

In a massive storage network, the first node that receives an object from outside always plays an important role in storing procedure, which includes routing, generating id and many other computing that need more resources. We do not start testing on this node because we still have enormous amount of nodes in our large-scale storage network. On the contrary, we just push the metadata identified by id to other ones with lower load. Note that we will use request to describe these information the algorithm need.

Actually, the request is send to functional bricks but not the nodes. We use functional bricks to build our system and each brick perform a certain part of the forwarding procedure. One physical node may host more than one functional bricks in parallel. There are two types of functional bricks: checking bricks and matching bricks. A checking brick can only handle a certain part of constrains assigned to a certain attribute and a matching brick can match the merged result produced by participating bricks with subscriptions. A request containing metadata and id will be send to several checking bricks one by one. Every checking brick checks if there is any attribute in metadata can be checked by self and send the remaining message to next brick. If such attribute exists, all related constrains will be calculated and produce a set of passed constrains, represented by some kind of Bloom filter. The result will be merged with those produced by other functional bricks. At last the final result will be send to a matching brick where the real forwarding addresses will be calculated.

4.2 Functional Brick

Functional bricks are basic computing units in our design.

4.2.1 Brick or node

First, limited to the scale of storage, one node may need to host several bricks to enforce our mechanism, though this should rarely happen since we are talking about large scale storage network.

Second, One physical node may have the capacity of hosting several functional bricks. For example, modern computing devices often contain several computing units in boxed, such as one or more GPUs and as well as multicore processors. It is likely to have the capacity of running several functional bricks at the same time.

Third, the functional brick actually defines the minimal size of computing that can be performed in parallel: different attributes can be handled at the same time and constrains share same attribute should be checked without being separated since many constrains are in partial order and most computing can be eliminated.

At last, another critical factor driving us to introduce the brick is that we can dynamically adjust the number of bricks to satisfy the fluctuant needs for computing resource and implement the system's self-tuning property.

4.2.2 A brick as a virtual machine

The idea of functional brick is inspired by system virtualization [15,16] and the architecture of Cloud Computing [17,18]. A functional brick is some kind of function-limited, lightweight virtual machine. Just like a virtual machine host in the cloud, a functional brick can be lunched from an image stored somewhere.

In our design, functional bricks are lunched using an executable functional image (FI), and a virtual storage image (SI) would be attached to it after launching. Both those images are stored in storage system just like other objects and can be fetched using their id. There are only several kinds of read-only standard functional images providing designed functions and many different storage images to store different runtime data. A functional image for checking brick provides programs to perform checking on a certain kind of data type while a functional image for matching brick provides binaries mapping the set of satisfied constrains to real world addresses. For example, a checking brick's functional image may be launched to a checking brick to perform all string constrains related to a certain attribute while an matching brick launched by a matching functional image can find out if there are any filter in the set being exactly satisfied and take out assigned addresses related to those filters in a subscription. Additional, the implement of function-specific functional image do not need to be unique in one storage system, there are maybe more than one implements to satisfy varying environment and a node can choose one of them to launch a new functional brick. A storage image contains information related to specific attribute, such as, a binary search tree contains all strings occurred in constrains related to attribute "producer".

With the combination of different functional image and storage image, a functional brick can act as different role. For example, a checking brick lunched from an functional image of handling strings and attaching a storage image which storing all constrains related to attribute "name" can check all predicates for attribute "name". When new subscription registers in system, new constrains will be merged into related virtual storage images.

Specific storage image usually have many copies to fasten the lunching time of functional bricks, but that may need additional mechanism to ensure the consistency of all copies on the other hand. Our suggestion is keeping few copies since the time used to launch is not so important and copying data to host's memory periodically to shorten the access time. The different copies using storage's duplicate mechanism to sync.

4.2.3 Self-tuning

An excellent advantage of using bricks is to make system self-tuning. For example, when too many forwarding requests occur in the storage network, more bricks can be launched to fit the need, and when fewer objects need forwarding we can terminate some of them to lighten system's load (then to reduce power consumption), or save resources to other tasks.

When a functional brick receives a request, first it will check whether it can handle such request, if not, it passes the request to other. Every request in the storage network with hold a number "hops" to record how many functional bricks have passed without being processed. If its "hops" reached a limited maximum value, a host can choose to launch a new corresponding functional brick for processing this request, if the physical host has resource to host another brick. If for a long predefined time a functional brick do not process any request, the brick will be terminated. Every functional brick keeps a queue for buffering and the queue's size should be carefully selected to trade-off the cost of transmission and the benefits of distribution.

4.3 Checking Constrains

A checking brick only handles predicates with a certain attribute, which calls its "target attribute", or "target string" since all attributes have type of string. Although checking bricks launched from different functional bricks have different behaviors, they still share some pattern in common. First, when received a message containing metadata, every checking brick need to find out if there any attribute can be test by self. Second, every checking brick enforce same mechanism to send remaining metadata to other checking bricks, to monitor subsequent checking bricks and merge self's result with others.

First, such a brick check if the request is a checking request. If not, the brick will pass it to another brick. Then it will find out if its target attribute is in the set of metadata's attributes. If not, the brick also pass it to another.

Second, the brick delete own target attribute and its value from metadata, and send remaining metadata to another brick at the same time. After this the brick set another thread to wait for the response.

Third, a brick use specific algorithm to calculate the result set (the result would be a Bloom filter to represent the set), which indicate those constrains satisfied by target attribute's value.

At last, when receives response from other brick, the brick update owns just calculated result with the received one and send response with new updated result to previous brick following the request's path. If not receiving the response for a limited time, the brick will send the remaining metadata to another neighbor.

4.3.1 Target attribute

When a request receives, one needs to quickly find out if there is any attribute that can be processed and pass the others to next. We use ternary search tree [19] to arrange the metadata. It takes O(logh) times to such in the tree, where h is the height of tree. Since all attributes are known when building the tree, we can build an optimal tree by, for example, sorting attributes and building by bisection method. Such a Tree performs much quicker for failed searching, and most of the searching would fail according to the relatively small size of the set of attributes for single object, so this should be quite fast.

4.3.2 Remote Bloom filter update

Bloom filters can effectively represents a set of all constrains that satisfied by an object. But the size is still too big to transmit. The simple method we used is to only transmit the non-0 bits. Although the total scale of the set containing all constrains is quite large, they are divided into different attributes and the number of constrains one object satisfied would not be too big. A functional brick holds self's Bloom filter and receives another Bloom filter with its non-0 bit, and set them to 1 on self's filter. We call this a method a remote bloom filter update. We use a general concept here since we are still seeking for a better algorithm.

4.3.3 Checking brick for numbers

For number attributes, we describe a quick checking algorithm as below.

For each attribute, we maintain an Lx4 table, where L is the total number of different values in all constrains related to this attribute. First column in the table records all values occurred in constrains, and the table has been sorted by this row. Every cell in Second to fourth column records a Bloom filter representing a set containing all constrains that would be

Number	>	<	=
-45.64	{5647, 7898}	{3611, 5564, 6617, 7118, 7130, 9142}	0
-38.26	{5647, 7898}	{3611, 5564, 6617, 7118, 7130, 9142}	0
-33.90	{2633, 5647, 6297, 7898}	{3611, 5564, 7118, 7130}	0
-26.93	{2632, 3005, 5010, 5647, 6297, 7898}	{3611, 5564, 7118, 7130}	{3540, 8132}
-17.08	{2632, 3005, 5010, 5647, 6297, 7898}	{3611, 5564, 7118, 7130}	0
-13.59	{2023, 2632, 3005, 5010, 5647, 6297, 7631, 7898}	{5564, 7130}	0
5.99	{2023, 2632, 3005, 5010, 5647, 6297, 7631, 7898}	{5564, 7130}	{3912, 8907}
14.50	{2023, 2632, 3005, 5010, 5647, 6297, 7631, 7898}	0	{2843, 8407}
20.93	{2023, 2632, 3005, 4189, 5010, 5647, 6297, 7631, 7898, 9473}	0	0
37.58	0	0	{8133, 9543}

Constrains	Bloom filter
{“>”, -45.64}	{5647 , 7898}
{“<”, -38.26}	{6617 , 9142}
{“>”, -33.90}	{2632 , 6297}
{“>”, -26.93}	{3005 , 5010}
{“=”, -26.93}	{3540 , 8132}
{“<”, -17.08}	{3611 , 7118}
{“>”, -13.59}	{2023 , 7631}
{“<”, 5.99}	{5564 , 7130}
{“=”, 5.99}	{3912 , 8907}
{“=”, 14.50}	{2843 , 8407}
{“>”, -20.93}	{4189 , 9473}
{“=”, -37.58}	{8133 , 9543}

Figure 1. Example of an Lx4 table maintained by numbers checking brick.

satisfied if this constrain is satisfied. We use F(i, op) to indicate the Bloom filter assigned constrain {op, v} in this table, where v is value in i-th row first column, and f(i, op) to indicate original Bloom filter of that constrain. Notes f(i, op) is all-0-Bloom filter if no such constrain exists. We fill the table using below rules,

$$F(i, \text{‘=’}) = f(i, \text{‘=’}), \quad (1)$$
$$F(0, \text{‘>’}) = f(0, \text{‘>’}), \quad (2)$$
$$F(i, \text{‘>’}) = F(i\text{-}1, \text{‘>’}) \text{ OR } f(i, \text{‘>’}), \quad (3)$$
$$F(L, \text{‘<‘}) = b(L, \text{‘<‘}), \quad (4)$$
$$F(i, \text{‘<‘}) = F(i+1, \text{‘<‘}) \text{ OR } f(i, \text{‘<‘}). \quad (5)$$

Take an example shown in Fig. 1, if we get a value 0.5. We could easily find the value −13.59 and 5.99, and the value in (5.99, <) indicates all satisfied constrain with an operation of “<” while the value in (-13.59, >) indicates those satisfied constrains of “<”. After applying bitwise OR operation on these values, we get Bloom filter {2023, 2632, 3005, 5010, 5564, 5647, 6297, 7130, 7631, 7898}. Such a Bloom filter represents a set of {{“>”, −13.59}, {“>”, −33.90}, {“>”, −26.93}, {“<”, 5.99}, {“>”, − 45.64}}, which is the set containing all constrains value 0.5 satisfied.

4.3.4 Checking brick for strings

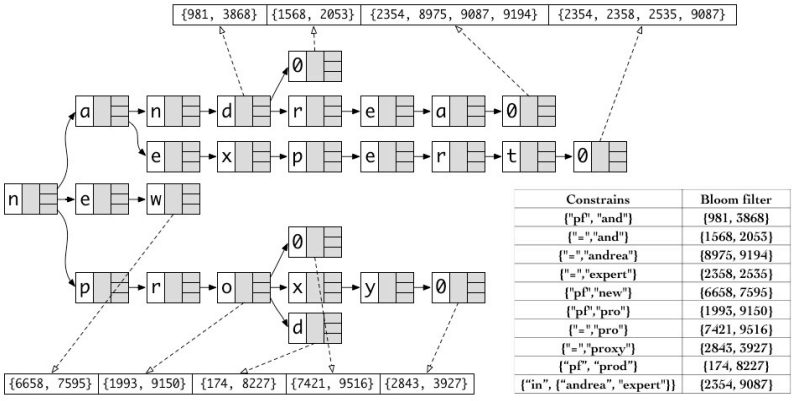
To keep simplicity, we only consider checking constrains of ‘=’, ‘PF’ and ‘IN’ operations, which can also be tested together.

For each attribute, we maintain a ternary search tree (TST) containing Bloom filters, whose each node represents a prefix of stored strings. All strings in the middle sub tree of a node start with that prefix. The string stored in TST contains the ending 0 which indicates the end of a string. If arrives in a node with the character ‘\0’, we would have matched a string exactly the same with the string it represented. For explanation, we call such a node 0 node, and others non-0 nodes. Each node in TST stores a character as well as an assigned Bloom filter. For 0 node the Bloom filter is the result of applying OR operations on constrain with “=” operation and all constrains with ‘in’ operations which string list contains this node’s represented string. On the other hand, for non-0 node, the Bloom filter represents constrain of ‘PF’ operation (0 if no such constrain).

When searching, the algorithm follows the path and update a Bloom filter whose bits are all 0 at first with each matched node’s Bloom filter. When finished, the final Bloom filter would already contain all constrains satisfied. Take the example shown in the Fig. 2 (Bloom filter of all bits are 0 is hided), if we get a string “andrea”, we can get the filter {981, 3868} and {2354, 8975, 9087, 9194} following searching path. And the final Bloom filter is {981, 2354, 3868, 8975, 9087, 9194}, or, the set of constrains {{“pf”, “and”}, {“in”, {“andrea”, “expert”}}, {“=”,”andrea”}}.

4.3.5 Monitoring chain

In our original design, we just send request to another node after finishing own computing on target string, containing the result this brick have just produced. After that,



Constrains	Bloom filter
{"pf", "and"}	{981, 3868}
{"=","and"}	{1568, 2053}
{"=","andrea"}	{8975, 9194}
{"=","expert"}	{2358, 2535}
{"pf","new"}	{6658, 7595}
{"pf","pro"}	{1993, 9150}
{"=","pro"}	{7421, 9516}
{"=","proxy"}	{2843, 3927}
{“pf”, “prod”}	{174, 8227}
{“in”, {“andrea”, "expert"}}	{2354, 9087}

Figure 2. Example of a ternary search tree contains Bloom filters

previous node does not care about anything about this object. When building the prototype, we found the problem soon. Our forwarding mechanism will be built on the massive storage network constituting large number of inexpensive commodity components. Due to the instability of such components, a message one node pushed to another may lose during the transmission, or the receiving node may crash and fail to finish the procedure. To resist the instability of components, we use monitoring chain to supervise the bricks involved. After pushing metadata to anther brick, the brick does not be terminated immediately, but remains a thread to wait for the response. This waiting uses few computing resources. If there is no response for a pre-set limited time. This brick will choose another brick to send request again. Now the question is: when to send the question?

One scheme is after processing. When having produced result, sending the metadata as well as the current checking result will reduce the network traffic since the response just need to contain a simple signal of having finished its work. By this way the last checking brick can also immediately send current result which is merged following path to matching brick and then push id to matched subscribers. But the disadvantage is inevitable: it may make users receive the id more then one times. For instance shown in Fig, when checking brick C2 not having received a response from C3, it does not know whether C3 have done its work and forwarded the request yet or not. So C2 may send request to another brick, said C5. If C3 has already finished its work and passed the remaining metadata to C4, C4 will continue the process since it does not know anything of previous bricks. For now there are two processes in whole system and those related subscribers would be likely to receive the id twice.

Another scheme is, on the contrary, sending the remaining metadata, if exists, to another brick before starting computing and waiting for response containing subsequent result. When receives, own result will be updated with the received Bloom filter and be send to previous brick after updating. Each brick only handle the first reply and ignore the others. By this way, we can ensure the id be sent to user only once. Fig. 4 shows the example.

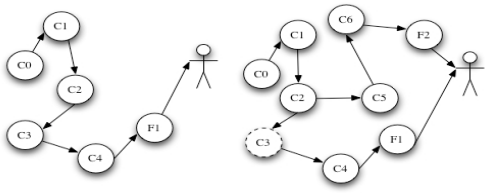


Figure 3. Failure example in scheme of no monitoring chains

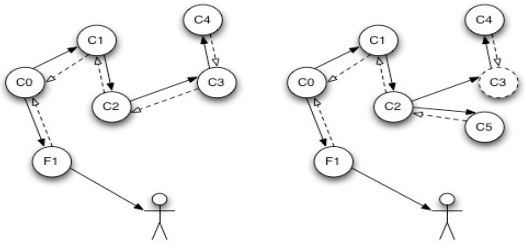


Figure 4. Failure example in scheme of no monitoring chain

We do not consider the situation that the head of monitoring chain fails because this brick hosted on the node that plays an important role in the storage procedure and failure occurred in this node may interrupt the storage procedure. Without having stored available copy of object, the forwarding result is senseless. So we leave such failure at chain's head to the underlay storage network.

4.4 Matching subscriptions

When a Bloom filter representing all satisfied constrains be produced, which would finally arrive the first brick started the algorithm, the Bloom filter and its id will be send to a matching brick for matching with all (or part of) subscriptions and find out the addresses the id should be send to. We say A matches B when A AND B equals to B, where AND is bitwise AND operation. The final matching procedure accepts a Bloom filter and a one-to-one list of addresses and its related Bloom filters and output the addresses that the id should be send to.

4.4.1 Subscription's Bloom filter

A subscription's Bloom filter represents a set containing all constrains that that subscription contains. That is, the result of feeding all constrains' Bloom filters to bitwise OR operation. For example, a subscription containing constrains which assigned Bloom filters are {1345, 7892}, {2363, 4654} and {2363, 9988} will be assign a Bloom filter {1345, 2363, 4654, 7892, 9988}, and Bloom filter {876, 1345, 1424, 2363, 4654, 7892, 9988, 9999} will match it.

4.4.2 Quick matching

We can check if A matches B using (NOT ((A AND B) XOR B)), where NOT, AND and XOR are bitwise operations. We do not use the bftree or sbstree used by Jerzak [20], because we think that (1) in our sceneries, there are many different subscriptions' Bloom filters can be matched. If we arrange these subscriptions into a tree, we need complex data structures and operations to travel among the nodes again and again. (2) This part of computing can be the most time-consuming part of whole procedure, but at the some time, is highly independent for each other and simple enough for just some bitwise operations. We can play a trick here: push this tasks to GPU, which can do these in parallel. We keep all subscriptions' Bloom filters in video memory to reduce the low-speed host to device copying and just send non-1 bit to GPU every time and output a list of wanted address's indexes.

4.5 Register subscriptions

The first thing for a subscribe/publish system is to register a subscription. The procedure is as below:

1) A subscription containing disjunction of conjunctions of elementary predicates will be separated to several

subscriptions containing only one filter (conjunction of predicates) with same target address.

2) For each one-filter-subscription, a Bloom filter will be assigned to every constrains in it, and the subscription's Bloom filter will be calculated as well.

3) New subscription's Bloom filter and its target address will be added to the list hosted by matching brick and will be merged into its storage image. And all constrains and their Bloom filters will be merged into corresponding attribute's structures in storage images.

5 Conclusion

It's necessary for a massive storage system to provide a content-based publish/subscribe service to support wide-area sensors network. Using this flexible object forwarding mechanism, data produced by wide variety of sensors can be aggregated and disseminated more effectively. By carefully using the limited computing resources of nodes in massive storage network, we proposed a scheme for this heavy task. Inspired by the development of system virtualization and the Cloud Computing, we introduced a concept of "brick" to dynamic schedule resources and tasks. Different functional bricks can be created from images to handle different attributes in a object, and at the same time they can be launched and terminated by actual demands and the resource situations of their host. We also provided same sample implements for checking brick of Integers and Strings. For every type of attribute, we have shown a fast constrains checking algorithm

6 Acknowledgements

The paper's work is supported by Jiangsu 973 project (No. BK2011022), Natural Science Foundation of China (No. 61170035) and NUST project (No. 2011YBXM18)

7 References

[1] P. Pietzuch, G. Muhl, and L. Fiege, "Distributed Event-Based Systems: An Emerging Community," Distributed Systems Online, IEEE, vol. 8, no. 2, p. 2, 2007.

[2] K. Sohraby and D. Minoli, Wireless sensor networks: technology, protocols, and applications. 2007.

[3] L. Tan and N. Wang, "Future Internet: The Internet of Things," in Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference, 2010, vol. 5.

[4] P. Costa and G. Picco, "Publish-subscribe on sensor networks: a semi-probabilistic approach," Mobile Adhoc and Sensor …, 2005.

[5] A. Carzaniga and C. P. Hall, "Content-based communication: a research agenda," in the 6th international workshop, New York, New York, USA, 2006, p. 2.

[6] R. Baldoni and A. Virgillito, "Distributed Event Routing in Publish/Subscribe Communication Systems: a Survey," Communications Surveys & Tutorials, IEEE, vol. 12, no. 1, pp. 39–58, 2010.

[7] L. Pamies-Juarez, P. Garcia-Lopez, and M. Sanchez-Artigas, "Enforcing fairness in P2P storage systems using asymmetric reciprocal exchanges," in Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on, 2011, pp. 122–131.

[8] Z. Zhang, Q. Lian, S. Lin, and E. al, "BitVault: A highly reliable distributed data retention platform," ACM SIGOPS Operating Systems Review, vol. 41, no. 2, 2007.

[9] H. Liu, S. Liu, X. Meng, C. Yang, and Y. Zhang, "LBVS: A Load Balancing Strategy for Virtual Storage," in Service Sciences (ICSS), 2010 International Conference on, 2010, pp. 257–262.

[10] A. L. Beberg and V. S. Pande, "Storage@home: Petascale Distributed Storage.," IPDPS, pp. 1–6, 2007.

[11] J. Kubiatowicz, D. Bindel, Y. Chen, S. E. Czerwinski, P. R. Eaton, D. Geels, R. Gummadi, S. C. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and Ben Y Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage.," ASPLOS, pp. 190–201, 2000.

[12] H. Weatherspoon and R. Rubin, "Persistence of Data in a Dynamic Network."

[13] D. Knuth, The art of computer programming. Volume 3, Sorting and searching. 1973.

[14] A. Carzaniga and A. L. Wolf, "Forwarding in a content-based network," in the 2003 conference, New York, New York, USA, 2003, ACM. pp. 163–174.

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, Xen and the art of virtualization, vol. 37, no. 5. New York, New York, USA: ACM, 2003, pp. 164–177.

[16] A. Kivity, Y. Kamay, D. Laor, and U. Lublin, "kvm: the Linux virtual machine monitor," in Proceedings of the Linux, 2007.

[17] R. Moreno-Vozmediano, R. Montero, and I. Llorente, "IaaS Cloud Architecture: From Virtualized Data Centers to Federated Cloud Infrastructures," Computer.

[18] S. Bhardwaj and L. Jain, "Cloud computing: A study of infrastructure as a service (IAAS)," International Journal of engineering and …, 2010.

[19] J. L. Bentley and R. Sedgewick, Fast algorithms for sorting and searching strings. Society for Industrial and Applied Mathematics, 1997, pp. 360–369.

[20] Z. Jerzak and C. Fetzer, "Bloom filter based routing for content-based publish/subscribe," in the second international conference, New York, New York, USA, 2008, pp. 71–81.

Parallel Algorithm for Building Extraction from LiDAR Data

Hyo Jong Lee
Div. of Computer Science and Engineering
Center for Advanced Image and Information Technology
Chonbuk National University, Jeonju, Korea

Abstract - *The data presented by Light Detection and Ranging (LiDAR) systems are in a dense and accurate three dimensional pattern without point classification, such as trees, roads, and buildings. Extraction of boundary points is essential for recognizing buildings. However, it is complicated to process the LiDAR data due to its irregularity and a large number of collected data points. In order to find boundary points in a quick and accurate way from a huge number of data points, a parallel algorithm for building extraction is proposed. In this paper, every processor reads the LiDAR data points and builds a quadtree respectively. Thus, a quadtree is built and shared through a network file system. Later, a breadth first search (BFS) is applied on every processor. The position of a node in the quadtree, in which has the number of children smaller than a predefined grain size, is stored in an array called as BFSArray. Process farming paradigm has been applied to process each nodes using MPI. In our primary experiment, results show significant speedup for multiple processors compared to a sequential algorithm.*

Keywords: Boundary point, process farming, MPI, LiDAR

1 Introduction

Recently *Light Detection And Ranging* (LiDAR) has become a reliable technique for 3D data collection [1]. The LiDAR system consists of three data collection tools: a laser scanner, Global Position System (GPS) for sensor position information, and an Inertial Measuring Unit (IMU) for orientation information. The laser scanner transmits and detects infrared signals to measure ranges.

LiDAR offers many advantages over traditional photo-grammetric methods for collecting elevation data. These include high vertical accuracy, fast data collection and processing, robust data sets with many possible products, and the ability to collect data in a wide range of conditions [2]. Therefore, LiDAR system has been introduced as a new tool that efficiently provides accurate data collection from extensive areas.

In LiDAR processing there are two completely different possible approaches for processing images. One is using 2-D images. The advantages of 2-D images are less time required for processing and less cost. However, images in 2-D format are inadequate for accurate modeling and for defining road boundaries and other building boundaries due to their lack of high resolutions [3]. Accordingly, recent demand and utilization of 3-D terrain information has been increased in various field and researchers have already investigated 3-D extraction methods of geographical features, buildings, and road networks [4].

Obviously more time is required for processing 3-D images. In order to find boundary point in a 3-D images format in a quick and accurate pattern, parallel computing algorithm is one option. In our paper, parallel computing algorithm is applied. Firstly, all LiDAR points are built in a quadtree structure in every processor. Hence, a shared quadtree is accomplished. Secondly, BFS search is used on every processor for finding all nodes, in which the number of children is smaller than a predefined grain size. Experiment results show that building a quadtree and BFS searching take only a small amount of time compared to finding boundary point. At this point, all preparation work for our parallel computing algorithm is finished. Afterwards processor farming is performed for distributing tasks to processors. Messages Passing Interface (MPI) is employed as a communication tool during the whole process. Details description will be presented later.

The rest of the paper is organized as follows. Section II introduces quadtree representation of LiDAR data and the concepts of processor farming model. In Section III we discuss the detail operation of our algorithm. Experiment results and discussion are described in Section IV. Conclusion is drawn in the last section.

2 Quadtrees and processor farming model

2.1 Quadtree and its properties

In the field of image processing, computer graphics, and remote sensing two dimensional point and ranging data are often indexed using quadtrees[5]. A quadtree is a tree data structure in which each internal node has up to four children. All data information are stored only in the leaf nodes. Data in a quadtree is often collected in point format, for example, a height measurement is made at that location, and represented by an (x,y,z) triplet with x and y representing coordinates and z representing a measurement at that location. Nevertheless, a parent node only has the information of the number of children and the area range of its all leaf nodes locate in.

Quadtrees are most often used as a representation of a regular partitioning of space where regions are split recursively into quadrants until there is only one point in each quadrant. In other words, each quadtree block (also referred to as a cell, or node) only contains one particle. In Fig. 1, the region is recursively divided into four parts if more than one particle exists inside a single block.

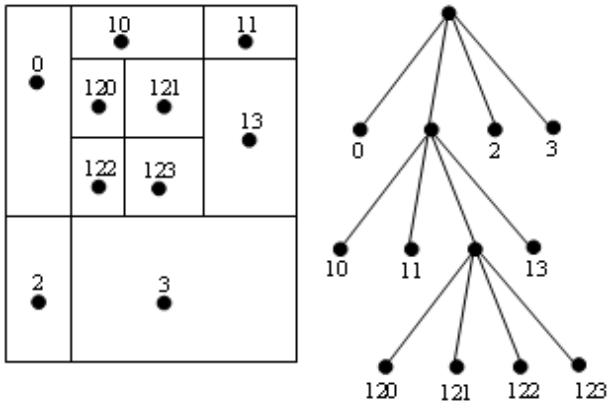


Figure 1. Quadtree structure

Quadtree-based spatial domain decomposition algorithm is designed for general use, and it can produce scalable geographical workload [6]. As we mentioned previously, LiDAR points are in a dense pattern, it is necessary to reduce the size of quadtree first before extracting boundary point. A quadtree with either a threshold or a maximum-depth limit allows us to reduce the size of tree at the cost of prediction accuracy, as it is no longer an exact copy of the original data. Setting the accurate threshold helps smooth the experimental data [7]. In this paper, a threshold value of height was set to compress a quadtree. A threshold means a criterion to merge any adjacent blocks as a single block if the height difference for the adjacent blocks is smaller than the predefined threshold.

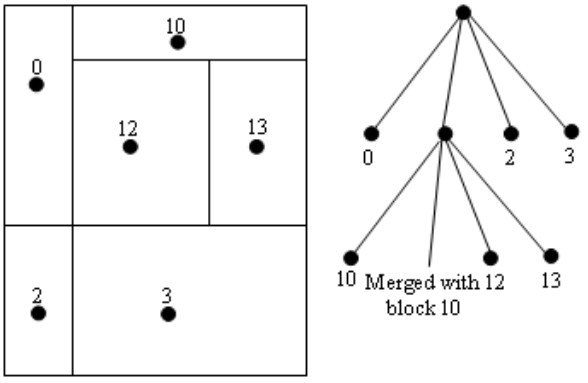


Figure 2. Reduced quadtree structure

Assuming the different height values in block 120, 121, 122, 123 in the Fig. 1 fall into the threshold, we can consider them as a single block and merge them. Equally, we assume the difference of height in block 10 and 11 is smaller than the threshold and the different height values in remaining blocks are bigger than the threshold. Therefore, block 10 and block 11 can be taken as a single block. After two block are merged, we use the average height as a new value to represent the height information in such block. Fig. 2 illustrates the reduced quadtree structure from Fig. 1 based on our assumption.

In this paper, the parallel computing algorithm is mainly focused. Therefore, finding the optimized threshold value for merging is out of scope of our problem. Adjusting the different threshold values will produce a more precise image.

2.2 Processor farming

The processor farming model works well even in a condition where pipeline structure cannot balance tasks or there are not enough buffers. Cok[8] also stated that efficient parallel algorithms could be implemented with processor farming.

Processor farming is defined as a group of independent tasks. The processors consist of a master processor and a number of worker processors [9]. A master processor controls the whole parallel processing. Our quadtree is not well balanced, that is, some nodes contain a little number of children, while other nodes contain a large number of children in BFSArray. It degrades the efficiency of algorithm if a master only sends one index of BFSArray to a worker processor at each time.

To solve the problem, we define a *lower bound* as grain size - α, and a *upper bound* as grain size + α at first. The α has been chosen to be equal to approximately 10% of the predefined grain size. When parallel computing algorithm begins, a master processor starts from the first index of the array. We call it a *start index*. If the number of children of a target node is smaller than a lower bound, then we move the pointer to the next index of array, until the total number of children of all nodes reaches lower bound or exceeds upper bound. Now, we mark the position of current pointer as an *end index*. If the second situation happens, we prefer to move the pointer back by one index. Surely, if the total number of children for all rest of nodes in BFArray is smaller than lower bound, then master sends the *start* index and the *last index* of BFSArray to a worker processor and broadcasts a "finish" signal to all worker processors. Worker processors stop receiving tasks as long as a "finish" signal received from master processor.

Likewise, a "finish" signal is sent back by a worker processor to the master processor as soon as it finished the assigned task, then the master processor repeats the previous procedure and sends *start* and *end* index to that worker processor until no more tasks are available in BFSArray.

Fig. 3 illustrates the first 13 elements of a hypothetical BFArray when grain size is defined as 40,000. The smaller number denotes the index of array. The number in the array box presents the number of children the node has. Actually, BFSArray is a pointer array and stores the address of those nodes and each node contains the information of number of children it has.

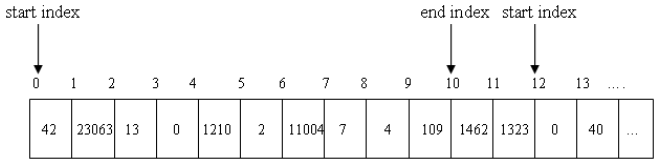


Figure 3. BFS array

When a grain size is 40,000, the lower bound and upper bound of grain size is 36,000 and 44,000, respectively. Consequently, a master processor stops moving a pointer to next index value if total number of children either falls between a lower bound and a upper bound or is beyond the upper bound scope.

The first element of BFSArray only has 42 children, and then master processor moves the index pointer to next one and calculates the summation of number of children. The index pointer stops at the index 10. Because the accumulated of children of the first eleven elements is 36916 $\in$ [*lowerbound; upperbound*]. After that, a master processor sends *start index* 0 and *end index* 10 to a worker processor. A master processor repeats this procedure until all tasks are assigned to each worker.

In next section, a detail operation for how the boundary points is generated by a proposal parallel computing algorithm will be discussed and our parallel computing algorithm will be presented.

3 Parallel Monotone Chain algorithm

Andrew's Monotone Chain algorithm [10] is implemented for extracting boundary points in this paper. The "Monotone Chain" algorithm computes upper and lower hulls of a monotone chain of points. It runs in O(*n*log *n*) time due to the sort time. After that, it only takes O(n) time to compute the hull, a hull has the same meaning of boundary point in our paper.

As we have introduced the LiDAR data previously, an object, such as a roof or a road, is represented by a set of dense data points. In order to extract the boundary points for an building, a reduction on quadtree size is preferred at first. Finding a group of neighbor points is a priori for extracting boundary points of an building. Without reducing the size of quadtree, it increases the computation time of finding groups of neighbor points. A group of neighbor points could be an object or multiple adjacent objects and the computation time for determining a group of neighbor points is $O(n^2)$. Because every point has to look for all other points for one time and the final neighbor points are determined. In our experiment, we find out reducing a quadtree does not take too much time. In contrast, looking for a group of neighbor points at the same height consumes most of computation time. Thus, reducing quadtree size is essential for speeding up the whole process.

Each index in BFSArray points to a node, in which has less number of children than a grain size. When a worker processor receives a *start* and *end* index of BFSArray, reducing each sub-quadtree according received index information. It starts to reduce the sub-quadtree, which is pointed by *start* index, and finishes until the sub-quadtree that is pointed by *end* index is compressed. After that, processor collects all the remaining data points in each sub-quadtree and sorts them by their height values.

Andrew's Monotone Chain is designed for 2D-image. In order to apply Andrew's Monotone Chain in 3D-image, we have to extract 3D-image points at different height level. Therefore, a reasonable threshold of 0.1 meter is used. When height threshold is used, all points with the height difference is smaller than the height threshold are considered on the same plane. With all the data points that a worker processor collects from reduced sub-quadtrees, a worker processor classifies all these data points. It starts from the data with the lowest height value and finds all other data points which are on the same plane. Afterwards, processors look for group of neighbor points on this plane. Finally, boundary points of each group of neighbor points can be found at this height level. After that, worker processor moves to next height level and find boundary point until all data points from reduced sub-quadtrees are searched.

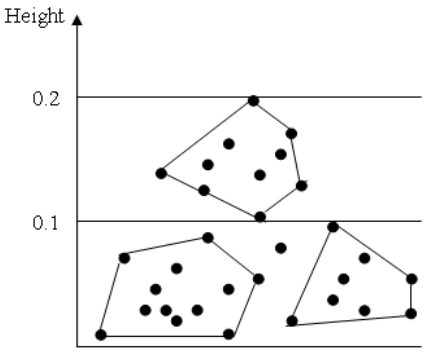


Figure 4. Finding boundary points

Suppose all the points in Fig. 4 are the remaining points after a processor performed a sub-quadtree reduction. Since all data points are sorted by height value, we know the data point whose height is the lowest. In Fig. 4, all height values have subtracted the lowest height value. Hence, the value of height starts from 0 in our example. Basically, we use the divide and conquer method to speed up the whole process. Instead of allowing one processor to find all groups of neighbor data points and then extract boundary point from each group of neighbor data points, we first store all LiDAR data points in a quadtree tree and master sends the index of BFSArray to processors, which can be considered as sending a small number of data in a small area each time to processors. With small amount of data points, worker processors finish the task much faster, since a huge amount of time on finding neighbor data points is saved. A master processor sends another task to a worker processor as soon as the worker processor finished previous assigned task.

Worker processors store the boundary point into a buffer and send the list of boundary points back to a master processor if buffer is full. We have not proved the optimal buffer size; however, the performance is acceptable in the

experiment when the buffer size is chosen as 4 or 5 times of grain size. When a master processor receives a boundary point list from a worker processor, then it resorts the all the boundary points and extracts a new list of boundary point as the same way as worker processor does.

Processor farming plays a key role in our parallel computing algorithm for extracting boundary point list. Fig. 5 presents the processor farming model of our parallel computing algorithm and the parallel computing algorithm for extracting boundary point is presented in Fig. 6.

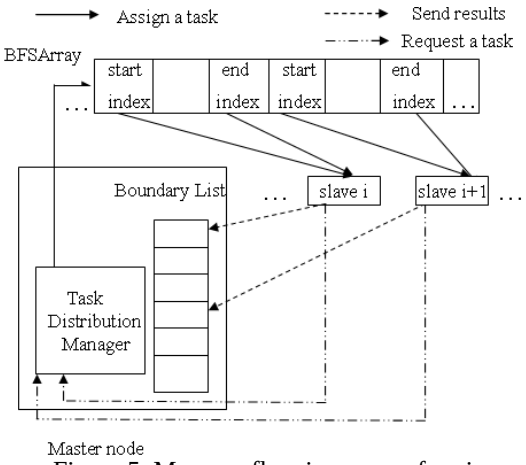


Figure 5. Message flow in process farming

Algorithm 1: Parallel Extraction of Boundary Points

***Master part*;**

Build a quadtree and creates a BFSArray;

Distribute a new task as soon as a worker finished an assigned task;

if *the buffer is full in a worker* **then**

Receives the list of boundary point in the buffer and resort all boundary points ;

end

Collect all boundary point list from workers when no available task in BFSArray;

Produce the final boundary point list;

***Worker part*;**

Build a quadtree and creates a BFSArray;

Receive a task and compresses sub-quadtrees;

Extract boundary point of buildings;

if *buffer is full* **then**

Sends a list of boundary points back to master;

end

Send the list of boundary points when all assigned tasks were finished;

Figure 6. Parallel algorithm for boundary extraction

4 Experiment results

The implemented parallel computing algorithm for abstracting boundary point was experimented on a clustering system, which has a master processor and 17 worker nodes. the master processor is Intel Xeon 2.8GHZ and each worker processor is Pentium 4 630, 3GHz. The parallel library is MPI 2.0 which was released by the MPI Forum [11]. The total number of data set is over four millions, which were collected by LiDAR from urban area.

Fig. 7 shows a sample LiDAR data and corresponding buildings extracted. Since LiDAR data are a set of coordinates with its height, the height values have been visualized as shown in Fig. 7 (a) using a rainbow bar. The extracted buildings were outlined connecting points from the Andrew's Monotone Chain algorithm. We confirmed visually that every building has been extracted correctly with an aerial photograph.

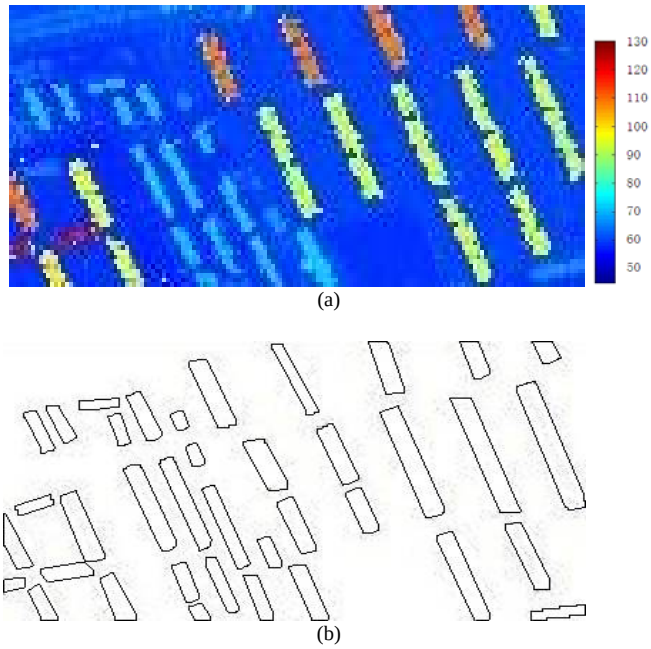


Figure 7. (a)Visualized sample LiDAR data (b) Corresponding buildings extracted

During the experiment, we tested the time for extracting boundary point with three different grain sizes. With over 4 million test data in this experiment, we chose the grain size 4,000, 40,000 and 400,000 respectively, and execution time is plotted in Fig. 8.

From Fig. 8, the performance is the best when the grain size is defined as 400,000, which is around 10% of all test data. When the grain size is small, the communication overheads increase. In our algorithm, worker processor takes much less time to finish an assigned task when the grain size is small. The reason is the complexity for finding neighbor points is $O(n^2)$. With less number of data points, the time for extraction of boundary points is much less. However, a master processor

takes more time for computation boundary point list from receiving worker processors and communication overheads were increased sharply.

From Fig. 8, it also indicates that scalability fails earlier when a very small grain size is applied. In our case, the scalability fails only after 3 processors involve, when grain size is chosen as 4,000, which is only 1/1000 of our test data. This issue should be taken care of in further study.

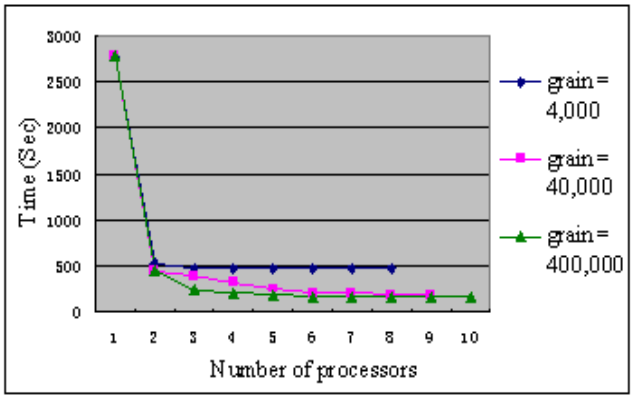


Figure 8. Excution time for extracting boundary points

5 Conclusions

Extraction of buildings from 3-D LiDAR data requires heavy computational processing. A parallel computing algorithm with processor farming model is proposed in this paper to reduce the processing time. Our algorithm is starting with building a quadtree on every processor and performs a BFS search. A BFSArray stores all the position of nodes, in which has less number of children than a predefine grain size. Processor farming distributes the indices of BFSArray to worker processors. The worker that completes a task first always asks for another task from the master. Boundary points only are sent among workers and master when the buffer is full in the workers or workers have finished all tasks in order to reduce the communication overheads. Grain size affects the performance of our parallel computing algorithm. With a small grain size, the scalability fails earlier and load balancing becomes an issue, since a master is busy at most time with extracting boundary points. The experiment results show our parallel computing algorithm improves the processing time six times faster than a sequential approach, which only one processor works.

In this paper, we do not focus on the accuracy of boundary points. However, with adjusting threshold or applying more sophisticated building extraction algorithms will generate more precise buildings. During the experiments, it proves that our parallel computing algorithm with a processor farming model works well for a large number of data set. Therefore, it is suitable for a vast range of parallel computing applications.

6 References

[1] N. A. Akel, O. Zilberstein and Y. Doytsher, "A Robust Method Used with Orthogonal Polynomials and Road Network for Automatic Terrain Surface Extraction From LiDAR Data in Urban Areas", The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Vol. 34, Part XXX., 2004.

[2] A. Rakusz, T. Lovas, and A. Barsi, "LiDAR-Based Vehicle Segmentation", International Archives of Photogrammetry Remote Sensing and Spatial Information Sciences, vol. 35, Part 2, pp. 156-159, 2004.

[3] Y. W. Choi, Y. W. Jang, H. J. Lee and G. S. Cho, "Heuristic Road Extraction", International Symposium on Information Technology Convergence, pp. 338-342, Nov. 2007.

[4] A. K. Shackelford and C. H. Davis, "Fully Automated Road Network Extraction from High-Resolution Satellite Multispectral Imagery", Proc. on IGARSS '03, vol. 1, pp. 461-463, July, 2003.

[5] H. Samet, *Applications of Spatial Data Structure*, Addison Wesley, Readings, MA, 1990.

[6] S. Wang and M. P. Armstrong, "A Quadtree Approach to Domain Decomposition for Spatial Interpolation in Grid Computing Enviroments", High Performance Computing with Geograogical Data, vol. 29, Issue 10, pp. 1481-1504, Oct. 2003.

[7] J. P. Grbovic, G. Bosilca, G. E. Fagg, T. Angskun and J. J. Dongarra, "MPI Collective Algorithm Selection and Quadtree Encoding", Parallel Computing, vol. 3, Issue 9, pp. 613-623, Sep. 2007.

[8] R. S. Cok, *Parallel Programs for the Transputer*, Prentice-Hall, 1986.

[9] H. J. Lee, and B. H. Lim, "Parallel Ray Tracing Using Processor Farming Model", Proc. on ICPPW '01, pp. 59-63, 2001.

[10] A. M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions", Inf. Proc. Letters 9, pp. 216-219, 1979.

[11] MPI Forum. MPI-2: *Extension to the Message-Passing Interface*. University of Tennessee, 1997.

Analog Fault Diagnosis based on S-Transform and PSO

A. Yanghong Tan [1,2], **B. Yichuang Sun** [1], C. **Yigang He**[2], **and D. Genmiao Zhang**[2]
[1] School of Engineering and Technology, University of Hertfordshire, Hatfield AL10 9AB, UK
[2] College of Electrical and Information Engineering, Hunan University, 410082, China

Abstract - *A novel method for fault diagnosis in analog circuits using S-transform (ST) as a preprocessor and Particle Swarm Optimization (PSO) based neural network (NN) identifier for classification is proposed in the paper. ST provides a frequency-dependent resolution and the features obtained from the ST are distinct, and easy to understand. The identifier, evolving with PSO, is a promising method to train NNs. It is faster and gets better results, and avoids some problems of conventional backpropagation (BP) algorithms. The comparison between the ST-based method and the wavelet-transform based method, and comparison between the ST and PSO method and the traditional NN method for analog fault diagnosis is provided. Simulation results show that the proposed method is effective in enhancing the efficiency of the training phase and the performance of the fault diagnostic system. The results clearly indicate higher correct classification of fault classes in the example circuits of similar faults.*

Keywords: analog fault diagnosis, PSO, S-transform, mixed-signal test, neural networks

1 Introduction

Analog circuit fault diagnosis has been an active area of research since the mid-1970s with significant work carried out at the system, board, and chip level [1]–[10]. A survey of the research conducted in this area clearly indicates that analog systems are among the most unreliable and least testable systems due to poor fault models, component tolerances, and nonlinearities compared with digital circuits [3-7]. This makes application of neural networks to this area very appealing since no mathematical model or comprehensive examination of these effects is required. The work in [2] is a pioneering effort, directly feeding samples of the impulse response of the linear circuits to the NNs as their inputs without any preprocessing. Nonlinear circuits are considered [3] in the frequency domain at different frequencies. Reference [4] presents a global parametric method, whose features are calculated from time domain response to a voltage step. It is executed with the use of an NN supplied with extracted features. The authors in [5] present a NN based method without preprocessing. The method requires a significant number of inputs even for a small circuit. Literature [7] presents a NN based system for linear circuit diagnosis, with preprocessing by wavelet decomposition, principal component analysis (PCA), and normalization to generate features for NNs. This technique is extended to actual circuits in [8]. References [9-10] deal with soft and multiple faults based on the parameter method. A wavelet NN method is presented in [11]. A method of designing specialized a-periodic excitation signal in time domain is described in [13], where single parametric faults are investigated by applying wavelet transform and genetic algorithms for further improvement.

Some outputs of a circuit are similar because of various factors such as element tolerance. In some cases, the outputs are so similar or even overlapped in their input space that we cannot successfully classify them. So they are often considered as ambiguous sets, which is not beneficial to real diagnosis applications. Therefore it is necessary to distinguish among fault classes with similar feature values, which always lead to a significant overlap. The work in [7] reveals that R6⇑ , R7⇓ , and R9⇑ (⇑ and ⇓ here indicate values higher and lower than the nominal values respectively in Fig.2) are similar faults, and a wavelet based method can successfully identify the faults of the circuit only if these similar faults are placed into one fault class [7]. Thus they are ambiguous sets. Otherwise, NNs cannot convergent at all.

The ST, as a hybrid of short-time Fourier transform (STFT) and wavelet transform (WT), has the similar form of STFT except its window width changing with frequency. So the progressive resolution can be provided by ST, just similar to WT. Moreover, ST interprets more straightforward frequency information than WT because a time-frequency axis is used in ST rather than a time-scale axis in WT. Hence ST provides a frequency-dependent resolution and the features obtained from ST are distinct, and immune to noise.

In this paper, we propose a S-transform based approach to analog fault diagnosis that overcomes the shortcomings of conventional methods. In this approach, we use ST as a processor to extract fault features. The outputs of a circuit under test are preprocessed by ST decomposition, PCA, and normalization to generate features. We adopt a PSO based NN identifier, which doesnot involve the gradient descent method and usually has fewer and less sensitive NN parameters than BP. This suggests that the PSO based NN identifier may be more robust.

2 S-Transform

Time-frequency analysis [13-16], as a tool of analyzing signals in both time and frequency domains simultaneously, is

very powerful in signal processing. Among the various proposed time-frequency analysis methods, STFT and WT are the most well-known and widely used linear transforms, which have the advantage of high efficiency and flexibility in the time-frequency filters. The progressive resolution can be provided by ST with its window width changing with frequency. And the ST, with a time-frequency axis is used rather than a time-scale axis in WT, is a more straightforward interpretation of frequency information than WT. So ST is a novel frequency-dependent resolution and the features extracted from ST are distinct and immune to noise.

The S-transform [13-16], is a powerful linear time-frequency representation. The filters in S-transform may consist of data adaptive weighting window with higher values for signal and lower for noise. The ST of x(t) [13] is

$$s(\tau, f) = \int_{-\infty}^{\infty} x(t) w(t-\tau, f) e^{-i2\pi ft} dt \tag{1}$$

Where *f*, *t* and τ are frequency, time and the center of the Gaussian window, ***w(t)***. ***w(t)*** is defined as

$$w(t-\tau, f) = \frac{|f|}{k\sqrt{2\pi}} e^{-\frac{f^2(t-\tau)^2}{2k^2}}, k > 0 \tag{2}$$

Where k is the dilation (window width) parameter, which determines the time frequency resolution. Unlike STFT, the width of window varies with frequency in S-transform. The S-spectrum of *x(t)* $\left(t \in [-\infty, \infty]\right)$ can be written as

$$\int_{-\infty}^{\infty} s(\tau, f) d\tau = \int_{-\infty}^{\infty} x(t) \int_{-\infty}^{\infty} w(t-\tau, f) d\tau e^{-i2\pi ft} dt \tag{3}$$

Letting T be the sample interval, the discrete realization of the ST can be calculated by taking the fast Fourier transform (FFT) and the convolution theorem:

$$s\left(jT, \frac{n}{NT}\right) = \sum_{p=0}^{N-1} X\left(\frac{m+n}{NT}\right) e^{-\frac{2\pi^2 m^2}{n^2}} e^{i\frac{2\pi m}{N} j} dt \tag{4}$$

Where $j, m, n = 0, 1, 2, \cdots N-1$ and N is the number of sampling points. $s(jT, n/NT)$ is available once the Fourier spectrum of $x(t)$ are obtained. And then compute the localizing Gaussian for the required frequency. We can get the S-spectrum by repeating the steps until all the rows of $s(n, j)$ to every frequency n have been obtained.

3 PSO based NN classifier

Conventional NNs have been widely used as trainable classifiers because of their generalization ability and function approximation [l8-21]. NN has the ability to form arbitrary nonlinear surfaces, and to respond consistently to data that are not trained with. However, the NNs use sharp decision boundaries to partition the feature space so their ability to estimate is limited. They are unable to correctly estimate class membership of data belonging to regions of the feature space if there is overlapping between the classes.

Particle Swarm Optimization (PSO) algorithm is a method for function optimization [18-20] based on social-psychological principles and provides insights into social behavior. The particles, which are a swarm of individuals, explore a multidimensional search space. The velocity vector is adjusted according to which particle goes through the search space. Therefore prior personal successful positions and the best position are found within a specific neighbourhood and act as attractors.

In PSO, a swarm of particles moves through a multi-dimensional search space [18-20]. A particle is defined by its current position *x* and velocity *v*. Each particle remembers the location y at which it has found the so far best solution to. The updating of velocity is

$$v(t+1) = wv(t) + c_1 r_1 (y - x) + c_2 r_2 (\hat{y} - x) \tag{5}$$

Where r_1 and r_2 are uniformly distributed random numbers from [0 1]. w is the inertia neural network weights, y is the personal best attraction potential. $\hat{y}$, the neighbourhood best position, is varied with the respective weights c_1 and c_2. The position $\hat{y}$ is either the best position of all particles (gbest) or the best positions within a local neighbourhood of the particle. The local neighbourhood (lbest) is to number the particles from 0 to $m-1$ and the neighbourhood of particle *i* consists of particle *i* itself and the two particles $i-1 \bmod m$. After the velocity has been updated, all particles move one step with their newly determined velocity as

$$x(t+1) = x(t) + v(t+1). \tag{6}$$

4 S-Transform and PSO based NN for Analog Fault Diagnosis

In this approach, the responses of the circuit under test (CUT) are simulated under the pre-assumed faults. And the sampled responses are processed by S-transform. The outputs of the ST are the so-called fault features. The features are applied to a BP neural network during the training phase. A unique code is given to a certain number of fault classes in advance. All the features and the associated fault codes are presented to a neural network as input–output pairs, when the BPNN is trained to its balance state after adjusting its weights and bias parameters. In the testing phase, we measure the responses of CUT stimulated by the same sources. Then the outputs of the ST are the features as input to the trained NN.

To investigate the effectiveness of the proposed approach in our fault diagnostic system, we use two circuits. The first real circuit is the Sallen–Key bandpass filter with a center frequency of 25 kHz [7], as shown in Fig.1.The nominal values for the components are also shown in Fig. 1. The tolerances of

the resistors and capacitors are assumed to be 5% and 10%, respectively. The faulty responses of the circuit are obtained when any of the components R3, C2, R2, and C1 is higher or lower than its nominal value by 50%, with the other three components varying within their tolerances. We consider the following fault classes: R3⇑ , R3⇓ , C2⇑ , C2⇑ , R2⇑ , R2⇑ , C1⇑ , and C1⇑ . The input source is an pulse of 5V peak and 10μs duration.

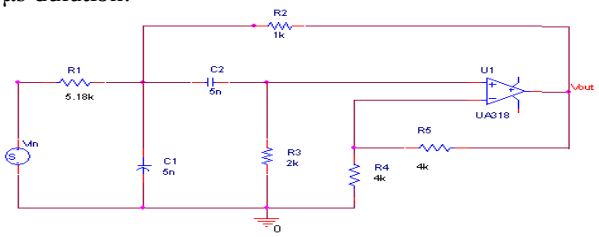


Fig. 1 25kHz Sallen–Key bandpass filter

The second circuit is a two-stage four operational amplifier (op-amp) biquad low-pass filter [7], shown in Fig.2. The authors in [7] pointed out that R6⇑ , R7⇓ , and R9⇑ are similar faults, and traditional fault diagnostic systems can successfully identify the sampled faults of the circuit only if these similar faults are placed together into one class of fault sets [7]. Otherwise, a neural network cannot be trained to distinguish among all the faults in [7].

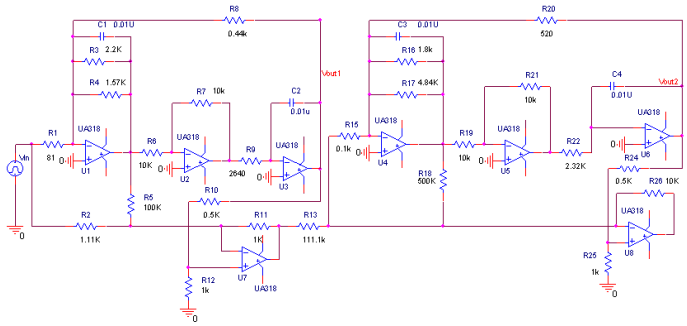
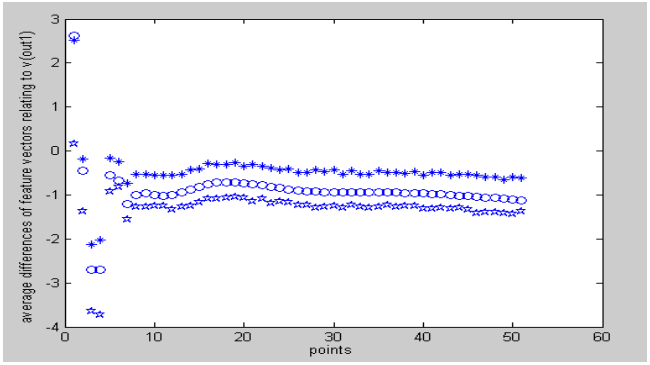
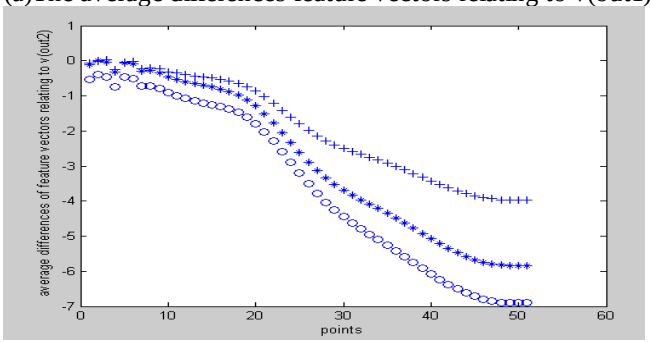


Fig.2. Two-stage 4-opamp biquad low-pass filter

Using our proposed approach, we can successfully classify all the faults, including the similar faults as mentioned above. By using ST processor, the difference of the contours for different faults becomes distinct. The features are obtained by principal component analysis and data normalization, after the S-transform of the responses of the circuits is available. The features are applied to NN and its outputs indicate the fault classes. The average difference feature vectors of 500 Monte-Carlo runs when considering the three faults R6⇑ , R7 ⇓ , and R9⇑ are presented in Fig.3, which shows that the differences are distinct.



(a)The average differences feature vectors relating to v(out1)



(b)The average differences feature vectors relating to v(out2)

Fig.3. The average differences of feature vectors of 500 Monte-Carlo runs when considering the three faults R6⇑ , R7 ⇓ , and R9⇑

(*: the average differences of feature vectors between R6⇑ and fault free

(o: the average differences of feature vectors between R7⇓ and fault free

(+: the average differences of feature vectors between R9⇑ and fault free)

To perform diagnosis for Fig.1, the work requires a two-layer BP network with 15 inputs, 10 neurons in layer 1, and 3 neurons in the output layer. The network was able to properly classify 100% of the test patterns.

The BP network for Fig.2 is with 50 inputs, 36 neurons in layer 1, and 5 neurons in the output layer. We put R6⇑ , R7⇓ , and R9⇑ in three different ambiguity groups. We input the feature vectors to the NN after sampling the circuit response, running S-transform, calculating the amplitude of the ST output, and conducting PCA and data normalization. Training and testing data for fault free are randomly generated using Monte-Carlo method. The data for fault free is obtained by varying the circuit components to their nominal values within their tolerances, and the data for fault classes are obtained by varying the circuit components to their faulty values respectively. For each fault pattern, we run Monte-Carlo analysis 1000 times, 500 for training data and the rest for testing data. Binary outputs are adopted here to classify the fault sets. The transfer functions of neural networks are a sigmoid function. The population size is 25, $c_1 = c_2 = 2$, the maximum value of $v(t)$ is 0.9, the minimum error is 0.0001,the range of inertia weights is [0.9,0.4], the ranges of NN weights

and bias for search are [-100,100] and [-8,8], respectively. We compare the proposed method with the conventional WT and BPNN method for analog fault diagnosis in the paper. The NNs are with the same structure and parameters. The results are shown in Table 1.

Table 1: comparison of the proposed technique and WT techniques

circuit	Accuracy based on WT(%)		Accuracy based on ST (%)	
	BPNN	PSO	BPNN	PSO
Fig.1	94.72	95.13	95.03	95.22
Fig.2 (case1)	training failure	training failure	89.54	91.47
Fig.2 (case2)	92.76	94.01	93.01	96.82

case1: R6⇑ , R7⇓ , and R9⇑ is considered as a fault set
case2: R6⇑ , R7⇓ , and R9⇑ is considered as three fault sets

Table 1 shows that the performance of our proposed technique is better than the wavelet based technique ([7]) for both circuits, especially for the circuit in Fig.2. We can correctly identify the faults R6⇑ , R7⇓ , and R9⇑ . However, the wavelet based technique cannot be trained to separate all the faults shown in Table I. The proposed approach leads to a higher correct classification of the test data for all circuits even when R6⇑ , R7⇓ , and R9⇑ are considered as three different fault sets.

5 Conclusions

We have developed a ST preprocessor and PSO based neural network identifier based analog fault diagnostic method. The ST, as a combination of STFT and WT, has the similar form of STFT with more straightforward frequency information than WT because a time-frequency axis is used in ST rather than a time-scale axis in WT. All the features are extracted from the output of ST. The method significantly enhances the efficiency of the training phase and the performance of the fault diagnostic system. Our results clearly demonstrate the superiority of the method, which leads to higher correct classification of fault classes in example circuits in the presence of similar faults. The PSO based NN identifier leads to higher correct classification.

6 Acknowledgements

This work was supported in part by the National Natural Science Foundation under Grant No.60876022, High-Tech Research and Development Program under Grant No. 2006AA04A104, National Natural Science Funds of China for Distinguished Young Scholar under Grant No. 50925727, by the Foundation of Hunan Provincial Science and Technology under Grant 07JJ6132 .

7 References

[1] Y. Sun ,Test and Diagnosis of Analogue, Mixed-signal and RF Integrated Circuits, IET Press, 2008

[2] H. Spence, "Automatic analog fault simulation," in Proc. Autotestcon Conf., pp. 17–22,1996.

[3] R. Spina and S. Upadhyaya, "Linear circuit fault diagnosis using neuromorphic analyzer," IEEE Trans. Circuits Syst. II, vol. 44, no. 3, pp. 188–196, Mar. 1997.

[4] P.Jantos, D. Grzechca, and J.Rutkowski, " A global parametric faults diagnosis with the use of artificial neural networks. " Proc. European Conference on Circuit Theory and Design, 2009,pp.651 – 654.

[5] M. Catelani and M. Gori, "On the application of neural networks to fault diagnosis of electronic analog circuits," Measurement, vol. 17, no. 2, pp. 73–80, Feb. 1996.

[6] P.Kyziol, D.Grzechca, and J.Rutkowski, "Multidimensional search space for catastrophic faults diagnosis in analog electronic circuits." Proc. European Conference on Cir. Theory and Design, 2009, pp:555 – 558.

[7] M. Aminian and F. Aminian, "Neural-network based analog-circuit fault diagnosis using wavelet transform as preprocessor," IEEE Trans. Circuits Syst. II, vol. 47, no. 2, pp. 151–156, Feb. 2000.

[8] F. Aminian,M. Aminian, and B. Collins, "Analog fault diagnosis of actual circuits using neural networks," IEEE Trans. Instrum. Meas., vol. 51,no. 3, pp. 544–550, Jun. 2002.

[9] M.Tadeusiewicz and S. Halgas, "An efficient method for simulation of multiple catastrophic faults", IEEE Int. Conf. Electron. Cir. Syst., ,2008,pp.356-359.

[10] M.Tadeusiewicz, S. Hałgas, and M. Korzybski, "An algorithm for soft-fault diagnosis of linear and nonlinear circuits," IEEE Trans.Cir. Syst. I, Nov. 2002. vol. 49, no. 11, pp. 1648–1653

[11] Y. He, Y. Tan, and Y. Sun, "Wavelet neural network approach for fault diagnosis of analogue circuits," Proc. Inst. Electr. Eng.-Circuits, Devices Syst., vol. 151, no. 4, pp. 379–384, Aug. 2004.

[12] C.Lukas, and R.Jerzy, "Specialised aperiodic excitation and wavelet transform improves analogue fault diagnosis. Circuit Theory and Design, ECCTD 2009. European Conference. PP.655 – 658.

[13] R.G.Stockwell, L. Mansinha, and R. P. Lowe, "Localization of the complex spectrum: the S Transform," IEEE Trans. Signal Process., vol.44, no. 4, pp. 998–1001, Apr. 1996.

[14] S. Santoso, E. J. Powers,W. M. Grady, and P. Hofmann, "Power quality assessment via wavelet transform analysis," IEEE Trans. Power Delivery, vol. 11, pp. 924–930, Apr. 1996.

[15] I. W. C. Lee and P. K. Dash. "S-Transform-Based Intelligent System for Classification of Power Quality Disturbance Signals. " IEEE Trans. on Industrial Electronics, Vol. 50, No. 4, pp. :800-805,August,2003

[16] M. V. Chilukuri and P. K. Dash, "Multiresolution Transform-based fuzzy recognition system for power quality events," IEEE Trans. Power Dev., vol. 19, no. 1, pp. 323–330, Jan. 2004.

[17] R. Carmassi, M. Catelani, G. Iuculano, etal, "Analog network testability measurement: A symbolic formulation approach," IEEE Trans. Instrum. Meas., vol. 40, pp. 930–935, Dec. 1991.

[18] J.Kennedy, and R. Eberhart,. "Particle Swarm Optimization". Proceedings of IEEE International Conference on Neural Networks. IV. pp. 1942-1948.1995.

[19] D. W. Boeringer and D. H. Werner, "Particle swarm optimization versus genetic algorithms for phased array synthesis", IEEE Trans. On Antennas and Propagation, vol.52, No.3, pp: 771-779, March 2004.

[20] T.O.Ting, M.V.C.Rao and C.K.Loo, "A novel approach for unit commitment problem via an effective hybrid particle swarm optimization", IEEE Trans. on Power Systems, vol.21, No.1, pp: 411-418, February 2006.

Multi-way Partitioning of Very Large Integrated Circuits

Jae Young Park, Soongyu Kwon, Kyu Han Kim, Hyeong Geon Lee, Jong Kang Park, and Jong Tae Kim
School of information communication engineering,
Sungkyunkwan University, Suwon, GyeongGi-Do, South Korea

Abstract *– This paper considers the problem of finding a minimum cost partitioning of a large logic circuit which is a collection of sub-circuits implementable with chips selected from a given library. Each chip in the library has a different price, area, and I/O pin. We propose a multi-way partitioning algorithm based on a simulated-annealing bi-partitioning. Chip set selection algorithm finds the sorted list of chip sets from minimum cost to maximum cost. Using the minimum cost chip set by chosen chip set selection algorithm, a multi-way partitioning algorithm will find a solution satisfied with area and pin constraints. We used MCNC benchmark circuits and the results satisfy the constrain conditions.*

Keywords: integrated circuit, MCNC benchmark, multi-way partitioning, simulated-annealing.

1 Introduction

As semiconductor's integration is increased, we can integrate many modules into only one circuit. However, it may not integrate into one circuit in real case because of size, cost or chip complexity. Therefore, we sometimes face with the situation that we must partition to several chips from one chip. In this reason, multi-way partitioning problem has become very important for many years ago. Goals for the partitioning are to make chip's partition satisfy area constraint and minimize interconnection between each chip. The reason to minimize interconnection between chips is that if the number of interconnections is increased, numbers of required pins are increased in each chip. This increases the size of entire system and degrades system efficiency.

Most of the previous partitioning algorithm has concentrated on heuristic method [1]. Partitioning algorithms using heuristic method need a proper initial solution, because heuristic methods start with temporary solution [2]. Having a better initial solution can make the final solution better. In this paper, we used low cost chip set selection algorithm that we proposed for initial solution and partitioning problem is proposed when variety of chips of different sizes are given in a library. We solved the multi-way partitioning problem using the simulated annealing bi-partitioning heuristic algorithm repeatedly with the low cost chip set selection algorithm [3].

In Chapter 2, we define basic definitions and explain the algorithm to find most suitable chip set. In Chapter 3, we explain multi-way partitioning algorithm. Next, Chapter 4 analyzes the results of the experiment. Finally, Chapter 5 is the conclusion.

2 Lowest Cost Chip Set Selection

This experiment defines partition size beforehand and process multi-way partitioning of different size. Here, we consider chip's cost according to its size so we decide on each block's size to materialize in a lowest cost.

Therefore, within greatest partition number, we find chip set with lowest cost. Then according to chips that makes up that chip set, multi-way partitioning is processed to get final partitioning result that satisfies each chip's pin limitation.

2.1 Basic Definition

(Definition 1): k number of partitioning, Pk = {C1, C2, … , Ck}, is done by k's clusters, C1, C2, … , Ck. At that instant, it gets relationship of $C1 \cup C2 \cup \ldots \cup Ck = V$. If k is 2, it becomes Bipartition problem.

(Definition 2)signal net, or net: A collection of pins which must be electrically connected [4].

Fig. 1 is an example of partitioning. If a netlist of n module V = {v1, v2, …, vn} are given, partitioning is set by already defined k number of clusters that each module is assigned [5]. Minimized objective function is defined as F(Pk) and this is partitioning result's cost function. Each cluster is assumed to be mutually exclusive.

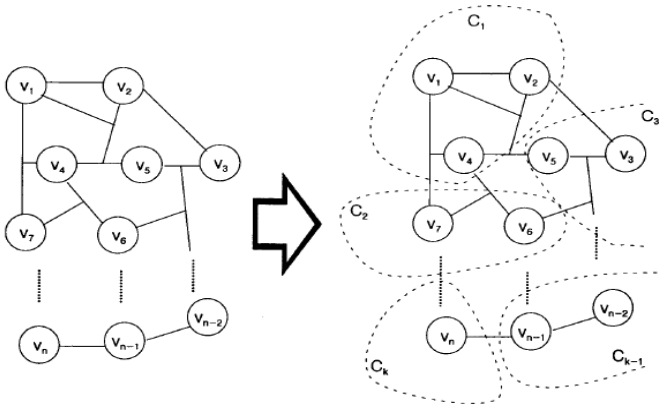


Figure 1 Example of partition to cluster

To express circuit's netlist in most regular way is hyper graph, H(V, E). Here, E = {e1, e2, … , em} is collection of signal net. Each e is collection of modules that connects this net. In all graphs, it is assumed that $e \in E$, $|e| \geq 2$. Weight function $w:V \rightarrow R$ is used in each module's area. If it is expanded to clusters, it becomes $W(C) = \Sigma v \in c w(v)$ and another weight function, $w':E \rightarrow R$ can be defined by net.

(Definition 3): For each module v, nets that are adjacent to v are $N(v) = \{e \in E | v \in e\}$, and collection of modules neighboring v is expressed as $M(v) = \{w \in V | \exists e\ v, w \in e, v \neq w\}$.

(Definition 4): Each signal net e is created by one source module S(e) and collection of destination modules D(e). This allows knowledge of direction of the signal.

(Definition 5): Collection of Hyper-edge Cut due to cluster c is expressed as in next formula. $E(c) = \{e \in E$ s.t $0 < |e \cap C| < |e|\}$. Here, at minimum, if some of pins in net e are in cluster c, relationship, $e \in E(c)$, is formed.

2.2 Problem Definition

Table 1 shows costs according to chip's area. As shown in Table 1, as chip's area increases, cost also increases greatly. Especially, when area is increased from 1(cm2) to 1.56(cm2), die cost and packaging cost in greatly increased so that it becomes noticeable. System made with 1.56(cm2) can be made with two 1(cm2) chips so looking at Table 1, system can be made with lower cost.

Most of the previous partitioning algorithm has concentrated on heuristic method [1]. Partitioning algorithms using heuristic method need a proper initial solution, because heuristic methods start with temporary solution [2]. Having a better initial solution can make the final solution better.

Table 1 Number of chipset according to partition

Maximum Partitioning Number	Number of Chipset
N=1	8
N=2	36
N=3	120
N=4	330
N=5	792
N=6	1716
…	…

2.3 Most Suitable Chip Set

There can be many chip sets depending on number of types of chips and number of chips that consist chip set. According to Table 2, if there can be 8 types of chips, number of chip sets available as much as greatest number of partition are collection that are with some duplication. Here, n is number of greatest partition.

As n gets larger, number of available collection increases. As n gets larger, system speed will get lower so n must be kept in reasonable value. This value made to be determined by the user's discretion. Next are assumptions for algorithms for selecting chip set. In net list, all cell's area is 1.

① Area between each chip's connection is ignored.

② Chip's decrease in speed is ignored from greatest partition number input from chip.tec file.

Each chip's factors, according to change in technology, like each chip's area, number of pins, and cost, are input from chip.tec file to get correct result. As situation changes, contents can be altered so we can get results from latest data. Greatest partition number is chosen by the user. This was done so that user can pick degree of system's decrease in speed due to partition.

Table 2 Cost for several die size[6]

Area (sq. cm)	Die /Wafer	Die yield /wafer	Cost of die	Cost to test die	Packaging costs
0.06	2778	79.72%	$0.25	$0.63	$5.25
0.25	656	57.60%	$1.46	$0.87	$5.25
0.56	274	36.86%	$5.45	$1.36	$5.25
1.00	143	22.50%	$17.09	$2.22	$5.25
1.56	84	13.71%	$47.76	$3.65	$52.25
2.25	53	8.52%	$121.80	$5.87	$52.25
3.06	35	5.45%	$288.34	$9.17	$52.25
4.00	23	3.60%	$664.25	$13.89	$52.25

2.4 Lowest Cost Chip Set Selection Algorithm

① All chip set $(d_j=(A_j,P_j,U_j)\ j=1,2,\ldots,m)$ are put in array – $D_{set}=\{d_1,d_2,d_3,\ldots,d_m\}$

② Eliminate all chip sets that has less area than Input circuit's area (A_{input}) from D_{set}.

③ Among remaining chip set $(d_j=\{ c_1,c_2,\ldots,c_k\})$, where k is 4. minimum, find lowest chip set's cost (U_{min}).

④ Eliminate all chip sets (d_j) from D_{set} where $U_{min} < ¼\ U_j$.

⑤ All remaining chip sets in D_{set} are arranged by cost.

⑥ Select lowest cost chipset (d_{min}) to be multi-way partition. When - $d_{min}= \{ c_1,c_2,\ldots,c_k\})$, all chips get condition $c_j=(a_j,p_j,u_j),\ (j=1,2,\ldots,k)$

While satisfying limitation of greatest partition number, to find lowest cost chip set, all chip sets that satisfies this condition have to get cost calculated (first step). For all the chip sets that are acquired through this, we get area (A) and cost (U). In second step, chip sets that have smaller area than input netlist get ignored. In third step, we find chip set that can materialize input netlist fastest that can use smallest partition. Its cost should be considered one of highest cost that can materialize input netlist. In fourth step, remaining chip sets are sorted by cost. Chip sets that are with higher cost than given cost are eliminated. Through this way, remaining chip sets are considered for materialization. In sixth step, chip set with lowest cost is applied multi-way partitioning algorithm to find partition that satisfies pin limitation. If it can't be found,

next lowest chip set is applied multi-way partitioning algorithm.

3 Multi-Way Partitioning Algorithm

In this thesis, problem that must be solved is multi-way partitioning problem. This problem must accept input of chips comprising of those satisfying given input netlist. It requires mostly at least three partitions. So we suggest algorithm that is altered appropriately from bi-partitioning algorithm to partitioning into number we want it to be partitioned.

3.1 Initial Partitioning

Basically, SA algorithm starts with temporary solution then finds best solution [7]. For this, initial solution is necessary. Information that needs to be input are partition number, each partition's greatest area, and each partition's greatest number of pins. Conditions for initial solution are partition number and each partition's area limitation. To satisfy these minimum conditions, given netlist is voluntarily partitioned. It can be formed into two ways like Fig. 2 depending how each block's cells are placed.

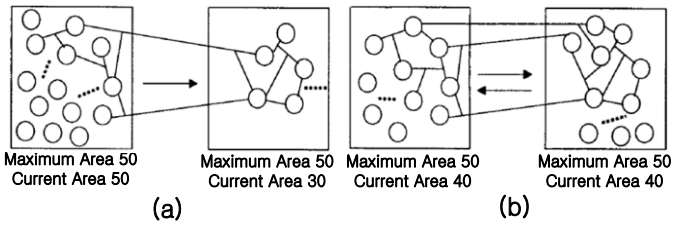


Figure 2 Initial partition

If given netlist's area is 80, and chips are partitioned into two chips with area of 50, like Fig. 2 (a), it can be partitioned so that most of cells can be in one block, or like (b), it can proportionally placed according to each chip's greatest area. In case of (a), in the beginning, cell's movement is limited but in case of (b), it's possible for easier movement. In case of (a), one side at greatest area limitation and the other is at lowest area limitation. Therefore, if cell from smaller side tries to move into larger side, it is limited. In this paper, for more free movement of cells, we applied method (b) for processing initial partitioning.

3.2 Selection of New Solution

After initial partition, each block that was partitioned must satisfy minimum and maximum area condition by moving one cell for new solution. Here, one cell is voluntarily chosen. However, in case if k partition, it can be moved to k-1 blocks. Here, requirements for new solution are lower number of cut set and high temperature even if number is high. Requirement for temperature may get lower as time passes but requirement for low number of cut set is in category of important condition. Therefore, selection of new solution is in direction where one cell moving through k-1 blocks with most reduction in cut sets. This would give increase in chance of accepting new solution in early stage in SA algorithm and lowers solution space that is created for multi-way partitioning. From each cell's movement point, k-1 movement direction's cost is each calculated and decide to go to most advantageous one. However, this decision does not guarantee that new solution is going to result in better results. Blocks that are earned when netlist is partitioned into k, each blocked is called A1, A2, A3, ..., Ak. If one cell from A1 group is moving to different group, it's direction is becomes A2, A3, ..., Ak. Here, cost change is defined as below. Here, cost is rise in cut set number.

$cel_{A1, A2}$: cost of A_1's cell is moving to group A_2

$cel_{A1, A3}$: cost of A_1's cell is moving to group A_3

$net_{A1, B1}$: rise of net B_1's cut set when one of A_1's cell is moving

3.3 Limitations That Final Solution Must Satisfy

Partitioning result that used SA algorithm have goal of reducing number of cut set (pin number) so final result would be a solution that have lowest number of cut set. However, if the result does not satisfy chip.tec file's pin limitation condition, we would not find solution that meets our demands. In this case, it comes to conclusion that it is impossible to find partition that fits input demands.

Since algorithm's goal is to minimize total number of pins, even if each partition satisfies pin limitation, it can move to solution that has smaller pin number but it does not satisfy pin limitation in partial blocks. So this means that in the end, it can reach solution that does not satisfy pin limitation.

Like Fig. 3, if netlist of area 758 is initially partitioned into two, pin number is each 634 and 357. While partitioning is continued, it meets solution that meets input requirements and pin limitation in the middle. However, partitioning still goes on so that at the end, it does not satisfy conditions. Of course, in total, it is possible to get a case with required pin number is 14, which is small, but it does not satisfy pin limitation of each chip. This thesis will solve this problem by continuously remembering solution with minimum number of pins that satisfies pin limitation during partitioning process. If final result does not satisfy pin limitation, output will be the solution that was memorized, which satisfies pin limitation. This case can be shown in result like Table 3 using t2 netlist.

This case can be shown in result like Table 3 using t2 netlist. As shown in Table 3, final result's block 4 exceeds pin limit of 120. Therefore, it is impossible to be materialized. However, result in the middle satisfies pin limitation. Program will maintain such result in the middle so when final result does not satisfy pin limitation, output will be the result in the middle.

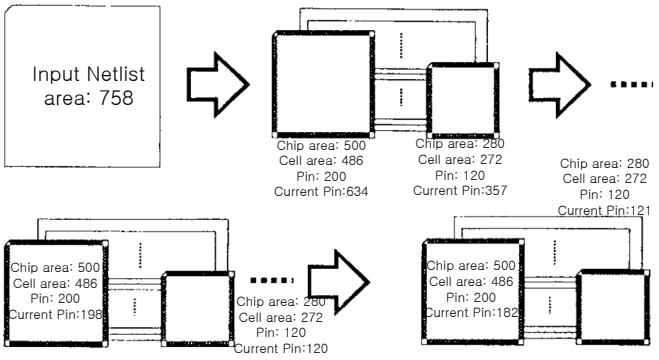


Figure 3 The relationship of the final solution with the final result

Table 3 Number of chipset according to partition

Block	Maximum Pin	Initial Pin	Middle of Process	Final result
Blk 0	200	891	93	93
Blk 1	120	633	88	86
Blk 2	120	672	34	34
Blk 3	120	614	50	51
Blk 4	120	696	120	121
		dissatisfaction	satisfaction	dissatisfaction

3.4 Simulated Annealing Algorithm

$$c(V_1, V_2) = |\{\{u,v\} \in E : u \in V_1 \ \& \ v \in V_2\}| + \alpha(|V_1|-|V_2|)^2 \quad (1)$$

$$c(V_1, V_2) = |\{\{u,v\} \in E : u \in V_1 \ \& \ v \in V_2\}| + \alpha * \sum f_i \quad (2)$$

Cost function applied to partitioning in SA[8], is equal to (1). Using this formula, during partition, as each block's area gets bigger, cost will get bigger. This would limit cell's free movement so it will downgrade solution's quality. In more improved version, SA[9], tried to fix this by making cost function to be (2). By making cell to freely move possible in between each block's area's Min and Max, final partitioning result becomes better. Fig. 4 is multi-way partitioning algorithm using simulated annealing. The most suitable chip set which is selected in previous level is partitioned to another chip set using the Fig. 4 algorithm.

4 Experimental Results

Program is written in C and circuit that was applied for this experiment is MCNC Benchmark circuits. Table 4 shows circuits that were used in the experiment. Most of circuits output results with first selected chip set but in case of industry2, it could not find satisfactory results from many chip sets.

Table 5 shows result when same information is used with maximum partitioning number of 4 and 6. Basically, as number increases, arrangement can be created with lower cost. Especially in case of t5.net, as greatest partitioning number is increased, cost is greatly reduced. However, as partitioning number is increased, circuit speed would decrease so it must be selected appropriately.

In case of industry2.net, selected chip set could not be partitioned so, many multi-way partitioning attempts occurred so this number would affect program process time. So when partitioning number is 6, unlike other cases, it would take less processing time.

1. Get a chip set .
2. Get an initial partition S .
3. Get an initial temperature T > 0 .
4. While not yet frozen do the following
   4.1 Perform the following loop L times.
      4.1.1 Pick a random neighbor S′ of S.
      4.1.2 Let $\Delta = cost(S') - cost(S)$
      4.1.3 If $\Delta \leq 0$ (downhill move)
            Set S = S′
      4.1.4 If S is satisfied with pin constraints
            Set S_{store} = S
      4.1.5 If $\Delta > 0$ (uphill move)
            Set S = S′ with probability $e^{-\Delta/T}$
   4.2 Set T=rT (reduce temperature)
5. If S is unsatisfied with pin constraints and S_{store} is satisfied with pin constraints
      Set S = S_{store} .
6. Return S .

Parameters :
L = the number of nodes * 16
cost = the number of crossing nets + size penalty
size penalty = $\alpha \times (|V_1| - |V_2|)^2$
$\alpha = 0.05$ $r = 0.95$

Figure 4 Simulated annealing algorithm[5]

5 Conclusion

The proposed algorithm is verified by C language with MCNC benchmark circuit. The C program is comprised of two big parts. First part is the algorithm that finds lowest cost chip set. Second part is algorithm that uses SA algorithm to do multi-way partitioning. In algorithm, input are net list file and library file. Outputs are each chip's area and result of partitioning that satisfies each chip's area and pin limitation. User inputs currently possible type of chip information through library to let program get information of each chip's size, pin number, and cost in given library. Program would find chip set with lowest cost with given information. That chip set would be input into multi-way partitioning algorithm. Multi-way partitioning used SA algorithm through new

neighbor structure. In process of multi-way partitioning, there could be result that satisfies pin limitation. It would be memorized so when final result does not satisfy pin limitation, even if it has more pins, result that satisfies pin limitation would be chosen as output. From partitioning algorithm, if result does not satisfy pin limitation, it is decided that given chip set cannot partitioned. Next lowest cost chip set would be then chosen for partition. It would repeat until chip set that satisfies pin limitation to get final result. Through experiment results, most got output that all chips' limitation is satisfied from first set of chip set. This result is partitioned system with lowest cost.

Table 4 MCNC benchmark circuits

Circuit	Pin Number	Net Number	Cell Number
p1	2908	902	833
p2	11219	3029	3014
t2	6134	1720	1663
t3	5807	1618	1607
t4	5975	1658	1515
t5	10076	2750	2595
t6	6638	1641	1752
19ks	10547	3282	2844
Industry2	48158	13419	12637
S15850P	24712	10383	10470
S38584P	55203	20717	20995
S9234P	14065	5844	5866
structP	5471	1920	1952

Table 5 Partitioning cost with maximum partition number

Circuit	Maximum Partition Number					
	4			6		
	cost	area	time	cost	Area	time
p1	39	840	11.7	39	840	26.9
p2	438	3030	65.1	306	3185	408.5
t2	94	1780	29.8	87	1745	107.8
t3	89	1625	31.7	78	1680	61.9
t4	80	1560	31.6	73	1525	63.4
t5	280	2625	66.7	143	2625	99.9
t6	94	1780	41.7	92	1900	59.7
19ks	368	2905	69.4	162	3000	118.5
Industry2	4918	12840	13617.4	4910	12650	13175.1
S15850P	4081	10560	706.0	3597	10590	631.5
S38584P	8162	21120	1375.4	8162	21120	1251.2
S9234P	1552	6120	176.4	1008	5905	272.9
structP	108	2000	32.9	101	1965	60.8

6 References

[1] Roman Kuznar, Rranc Brglez, and Krzysztof Kozminski, "Cost Minimization of Partitions into Multiple Devices", 30th ACM/IEEE Design Automation Conference

[2] S. Kirkpatrick, C. D. Gelatt, Jr., M. P. Vecchi, "Optimization by Simulated Annealing", Science, 220(4598), pp. 498–516, May 1983

[3] J. Y. Park, S. Kwon, K. H. Kim, H. G. Lee, J. T. Kim "Low Cost Chip Set Selection Algorithm for Multi-way Partitioning of Digital System", ICCESSE 2012

[4] Sadiq M. Sait and Habib Youssef, "VLSI PHYSICAL DESIGN AUTOMATION", IEEE PRESS, pp.28

[5] Charles J. Alpert and Andrew B. kahng, "Recent Directions in Netlist Partitioning: A survey" Integration : the VLSI Journal, 19(1-2), 1995, 1-81

[6] John L Hennessy and David A Patterson, "Computer Architecture A Quantitative Approach," , pp. 62.

[7] S. Kirkpatrick, C. D. Gelatt, Jr., M. P. Vecchi, "Optimization by Simulated Annealing", Science, 220(4598), pp. 498–516, May 1983.

[8] David S. Johnson, Cecilia R. Aragon, Lyle A. Mcgeoch, Catherine Schevon "Optimization By Simulated Annealin: An Experimental Evaluation; PART 1, GRAPH. PARTITIONING" Operation Research Vol. 37, No. 6, November-December 1989.

[9] Ching-Wei Yeh, Chung-Kuan Cheng and Ting-Ting Y.Lin "An Experimental Evaluation Of Partitioning Algoritms" IEEE Int. ASIC conference, 1991, pp14

Study on Parallel Compressed Sensing for Mass Data in Internet of Things

Yongping Zhang, Gongxuan Zhang, Yongli Wang, Zhaomeng Zhu, and Wei Zhang
School of Computer Science and Technology, Nanjing University of Science and Technology, Nanjing, Jiangsu, China, 210094

Abstract - *Internet of Things is the network of interconnection between people and things, and between things themselves by embedding additional gadgets, such as sensors, RFID tags. Mass data are usually collected, transmitted, processed and stored in Internet of Things. In this paper, a novel sampling method, i.e. compressed sensing, is used in processing mass data in Internet of Things. Compressed sensing can significantly lower the network communication burden by reducing sampling rates of sensors, but it is nonadaptive and its algorithm is high computational complexity. For this reason, we introduce redundant dictionary into compressed sensing for increasing the flexibility and put forward the idea that is parallel processing of compressed sensing algorithm in order to improve computing speed in this paper. In the latter part of this paper, we describe the framework of parallel compressed sensing algorithm and point out our current experimental results and future work.*

Keywords: Internet of Things; compressed sensing; parallel processing; optimal recovery; adaptive

1 Introduction

The term Internet of Things (IoT), also known as Web of Things, was first used in 1999 [1]. In 2005, *The Internet of Things* [2] of ITU reported that: the Internet of Things is "*any time, any place, and any things connection*" and "*ubiquitous networks, ubiquitous computing*". In brief, Internet of Things is the Internet of things to things by embedding intellisense system, such as RFID [3], sensor technologies. In Internet of Things, at first, the information about things are acquired by intellisense system; and then the data will be sent to data processing server through a variety of transmission networks; the next, these data are processed in the data processing server; as a result, the interactions of Human to Human, Human to Thing and Thing to Thing will be implemented [2]. This is similar to wireless sensor network (WSN) [4], and somebody think that Internet of things is wireless sensor network, but they are different, such as network architecture [5], [6].

Generally, the wireless sensor network protocol consists of five layers: application layer, transport layer, network layer, data-link layer, and physical layer; but the structure of Internet of Things can be classified into three layers [7] as shown in Figure 1. Perception layer can identify and acquire information of things; the primary function of transport layer is data transmission by various communication networks; and sundry different applications can be achieve in application layer.

Application Layer	Industrial Monitoring	Smart Home	Traffic Monitoring
	Mine Safety	Environmental Monitoring	… …
Transport Layer	Internet	Information Center of IoT	Private Network
	Mobile Network	Management Center of IoT	… …
Perception Layer	Bar Code Reader	Sensors	Intelligent Terminal
	RFID Sensors	Access Network	… …

Figure 1. The structure of Internet of Things

Internet of Things is a popular scientific and technical terminology and it has been classified as national key technology in many countries. Internet of Things can be used for various application areas, such as mine safety, smart home, forest fire detection, traffic monitoring and logistics monitoring. Figure 2 shows the examples of application areas of Internet of Things.

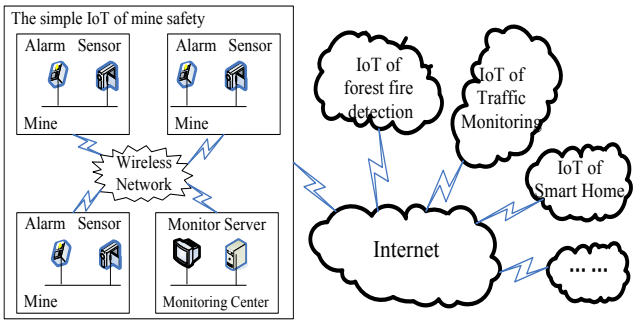


Figure 2. The applications of Internet of Things

There are usually mass data in Internet of Things. For example, a large-scale intelligent traffic or real-time ecological monitoring system, the amount of data generated can be achieved TB every day and PB each year. The existence of mass data would seriously affect sensors life,

Internet of Things coverage, and data processing and storage capacity of Internet of Things. To reduce the amount of data in Internet of Things, compressed sensing that proposed by Donoho [8], [9] and Candès [10], [11] is a good choice. Compressed sensing can significantly lower the network communication burden by reducing sampling rates of sensors, but it is nonadaptive and its algorithm is high computational complexity. So, it is necessary that parallel computing [12], [13] of compressed sensing algorithm for improving computing speed and redundant dictionary be introduced into compressed sensing for increasing the flexibility.

The remainder of this paper is organized as follows: In Section 2, we introduce compressed sensing theory. A new compressed sensing algorithm framework is the subject of Section 3. In the end, the conclusions and future study are given in Section 4.

2 Compressed sensing

Compressed sensing or compressive sampling (CS) is a new sampling mean that data sampling and compressing can be done simultaneously. Compressed sensing goes against the common principle (i.e. Nyquist density sampling theory), and it can sample at a low rate and later reconstruct the full-length signal based on numerical optimization algorithm [11], [14].

2.1 Compressed sensing theory

We suppose that there are an one-dimensional linear vector $X_{N\times 1}$ (which can be signal, image, etc) and a sparse transformation basis $\Psi_{N\times N}=[\psi_i,\ i=1,\ 2,\ldots,\ N]$. The transformation basis are assumed to be orthonormal basis at the beginning of compressed sensing be proposed. Then, we can obtain the conversion coefficient Θ about vector X based on transformation basis Ψ:

$$\Theta=\Psi^T X \qquad (1)$$

where $\Theta= [\theta_i=<X,\ \psi_i>=\psi_i^T X,\ i=1,\ 2,\ldots,\ N]$ is sparse. Here, the sparse means that the vast majority entries of Θ are zero [15].

The design of measurement matrix $\Phi_{M\times N}$ $(M<<N)$ is the next work. Φ must be independent of the transformation basis Ψ. Observation data Y about vector X would be obtained from the following equations:

$$Y=\Phi\Theta=\Phi\Psi^T X \qquad (2)$$

where Y is the compressed data. Matrix representation of the equations *(2)* is shown in Figure 3.

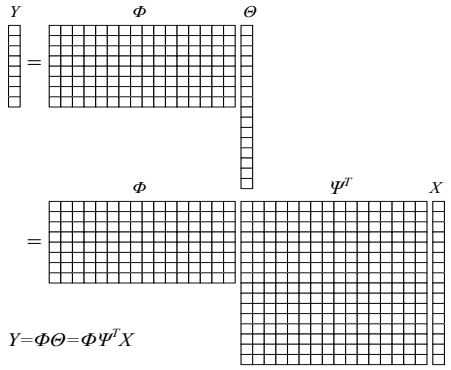


Figure 3. Matrix representation of observation vector

According to compressed sensing theory, we can reconstruct original signal X from the observation data Y via solving the 0-norm optimization problem:

$$\min \|\Theta\|_0 \qquad (3)$$
$$\text{s.t. } \Phi\Theta=\Phi\Psi^T X= Y$$

where $\|\Theta\|_0$ is the number of nonzero entries in Θ.

The computing process of compressed sensing is shown in Figure 4:

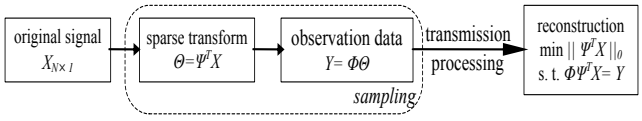


Figure 4. Theoretical framework of compressed sensing

2.2 Critical issue

The compressed sensing theory involves three key issues [16], [17]:

2.2.1 Sparse representation

Sparse representations of signal can be attributed to design of transformation basis. The transformation basis is orthonormal basis at the beginning of compressed sensing be proposed and it is nonadaptive. Peyré expand the requirement of transformation basis to orthogonal basis dictionary [18] that is formed by multiple orthogonal bases, and it can be adaptive to find an optimal orthogonal basis even unknown any information about signal in the dictionary.

2.2.2 Measurement matrix

The equations *(2)* are underdetermined and generally we cannot solve it. But if we suppose that $\Phi\Psi^T$ obey the restricted isometry property (RIP) [19], the equations *(2)* can be solved either exactly or accurately. The necessary and sufficient condition of RIP is transformation basis Ψ independent of measurement matrix Φ.

Gaussian stochastic matrix [20] is independent of most of matrix composed of orthogonal basis and it can act

as measurement matrix. In addition to Gaussian stochastic matrix, Noiselet matrix [21] and Rademacher stochastic matrix [22] obey the RIP too. But now, any measurement matrix can not guarantee 100% accurate reconstruction signal, the design of measurement matrix is still a hot topic of research.

2.2.3 Reconstruction

Accurate and exact reconstruction signal, looking for exact or approximate object $\hat{X}$ having coefficients with smallest 0-norm from the expression *(3)*, is a nonlinear optimization problem. We almost can not solve it because its solution is NP-hard [20]. However, a new theory was proposed by Donoho, Chen and Saunders [23]: when measurement matrix Φ was independent of transformation basis Ψ, we can solve the expression *(4)* instead of considering the expression of *(3)*:

$$\min \| \Theta \|_1 \quad \text{s.t. } \Phi\Theta=\Phi\Psi^T X= Y \qquad (4)$$

where the expression of *(4)* is a convex optimization problem and we can solve it as a linear programming problem.

2.3 Reconstruction algorithm

Currently, the compressed sensing reconstruction algorithms can be grouped under three categories [24]: greedy algorithm, convex relaxation method, combination algorithm.

2.3.1 Greedy algorithm

Greedy algorithm is iterative algorithm. It can gradually approache the original signal by selecting a local optimal solution in each iteration. This family of greedy algorithm includes Matching Pursuit (MP) [25], Orthogonal Matching Pursuit (OMP) [26], Stagewise Orthogonal Matching Pursuit (StOMP) [27] and Regularized Orthogonal Matching Pursuit (ROMP) [28], etc.

2.3.2 Convex relaxation method

Convex relaxation method is linear programming method for minimization 1-norm and it can solve original signal by transforming non-convex problem into a convex problem. This family of convex relaxation method includes Basis Pursuit (BP) [23], Gradient Projection for Sparse Reconstruction (GPSR) [29], Iterative Thresholding (IT) [30], Bregman Iterative (BI) [31], etc.

2.3.3 Combination algorithm

Combination algorithm can quickly reconstruct original signal by group testing. This family of Combination algorithm includes Fourier representations [32], Chaining Pursuit (CP) [33], Heavy Hitters on Steroids Pursuit (HHSP) [34], etc.

3 Parallel compressed sensing

In this section, we will describe a new compressed sensing algorithm framework and its technical details.

3.1 Parallel compressed sensing algorithm framework

Compressed sensing is a sampling method to achieve data compression while data is sampled, and the method can significantly reduce the amount of sample data in Internet of Things. However, everything has two sides; compressed sensing algorithm has also many inadequacies. For example, the transformation basis is nonadaptive, it cannot adapt the change of the signal types; compressed sensing algorithm is high computational complexity, especially the reconstruction algorithm. Such as BP algorithm, when length of signals is 8192, calculation scale of reconstruction is equivalent to solving linear programming problem with 8192×262144 [23].

According to our study, we propose an improved algorithm framework as shown in Figure 5:

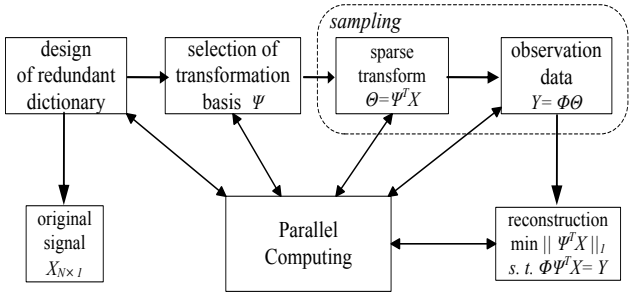


Figure 5. Theoretical framework of parallel compressed sensing

Compared with original compressed sensing algorithm, the new algorithm adds tow steps of redundant dictionary construction and transformation basis selection, and parallel computing is introduced into the new algorithm framework. In the section 3.2, we will further describe the parallel algorithm framework in detail.

3.2 Detailed description of parallel algorithm framework

3.2.1 Parallel computing of compressed sensing

Parallel computing [35] is a form of computation and carries out many calculations simultaneously. The principle of parallel computing is that many large problems can be divided into smaller parts. High-performance computing [35] is often regarded as parallel computing, because it cannot do without the support of parallel processing. Now, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors. In general, multicore parallel processing can select CPU programming or GPU programming [13].

Recently, the development of hybrid systems on the CPU and GPU is attracting our more and more attention.

Compressed sensing theory involves a large number of matrix operations, such as selection of transformation basis, sparse representation of signal, data compression and reconstruction signal. This facilitates the parallel processing of compressed sensing algorithm. And compressed sensing algorithm is high computational complexity, especially the reconstruction algorithm. So, parallel processing of compressed sensing algorithm is necessary and our study will focus on it. Parallel processing can involve in all the operations of compressed sensing algorithm, such as redundant dictionary construction, transformation basis selection, sparse representations, observation data obtainment, and reconstruction, would be implemented in the parallel algorithm framework.

3.2.2 Redundant dictionary

Sparse representation of signal is premise and base of the compressed sensing applications and it is determined by the transformation basis. We hope that the representation can adapt the change of the signal types but orthonormal basis is nonadaptive in original compressed sensing algorithm and the orthogonal basis dictionary cannot solve the problem.

Our selection about transformation basis is redundant dictionary in the parallel algorithm framework. Sparse representations that are based on redundant dictionary [36] are new theory of signal representation. In redundant dictionary, traditional orthogonal basis functions are replaced by over-complete redundancy function system. The use of redundant dictionary increases the flexibility of signal sparse representation, and we can choose transformation basis depending on the different needs. Now, we are studying the construction algorithm of redundant dictionary to fit sensors. In general, the design of redundant dictionary needs to spend a lot of computing and its parallel processing would be study.

3.2.3 Measurement matrix and observation data

According to Section Ⅱ, Ψ must be independent of Φ in compressed sensing. Stochastic matrix, such as Gaussian stochastic matrix, Rademacher stochastic matrix, is widely used as measurement matrix Φ. However, in practice, it is difficult to apply in large-scale problems because of the high computational complexity of stochastic matrix.

In compressed sensing theory, data compression is to calculate the observation data Y by the equations *(2)*. In order to fast to process mass data, it is important to construct adaptive measurement matrix based on sampling mechanism of sensor and design fast calculation method of the equations *(2)*. Parallel processing, clearly, can provide support for these.

3.2.4 Reconstruction algorithm

All compressed sensing reconstruction algorithms can be grouped under three categories: greedy algorithm, convex relaxation, combination algorithm. Parallel processing of the reconstruction algorithm can improve the calculation speed, reduce the execution time, and enhance real-time. In our experiments, under the same conditions, the reconstruction algorithm used dual-core CPU than single-core CPU can save the execution time of approximately 40% and 4-core CPU than single-core CPU can save the execution time more than 70% (shown in Figure 6), then we think that parallel reconstruction algorithm is feasible. The experiments of GPU parallel computing have not been completed and we are working on it. In addition, the hybrid systems on the CPU and GPU and cloud computing platform are also worth trying.

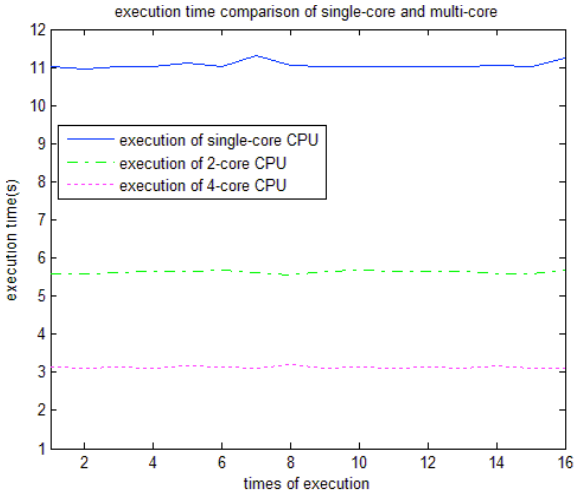


Figure 6. Execution time of OMP algorithm

According to our experience and studies, combination algorithms are more suitable for parallel computing in the three families of algorithms. The next work, we will focus on studying the mainstream combination algorithms, finding suitable algorithms to handle mass data and improving appropriately these algorithms. In the last, one of the most appropriate algorithms will be selected and achieved parallel processing.

3.2.5 Sensors

In our parallel algorithm framework, the terminal sensors were not concerned. As the basic data acquisition equipment, improving sensors performance will greatly promote the development of Internet of Things because the computing power and storage capacity of sensors are so limited. We have two ideas to improve the end of data collection:

1) Compressed sensing sensors: The compressed sensing sensors can be classified according to the application environment. Different sensors configure different simple redundant dictionary, and these redundant

dictionary can only adapt to limited choices. Sensors can compress sample data by compressed sensing system and send compressed data to backend server. Backend server can support parallel computing and any compressed sensing algorithms about all redundant dictionaries. In this way, the data transmission burden can reduced and computing speed can be improved.

2) Other sensors: Data processing platform of compressed sensing is added near the sensors. Each data processing platform can receive sample data from its neighboring multiple sensors. The sample data are compressed on data processing platform, and then transferred to the backend server. The server can support parallel computing and any compressed sensing algorithms about all redundant dictionaries, too.

4 Conclusions

In this paper, we try to process mass data in Internet of Things by using the compressed sensing theory, and propose a new idea that parallel computing of compressed sensing algorithm. In order to improve the adaptability of compressed sensing, redundant dictionary is added in the parallel compressed sensing algorithm framework. But we have not implemented a complete algorithm, which will be our next study work. In addition, it is a good research that compressed sensing algorithm be done on cloud computing platform and we will try it.

5 Acknowledgment

The paper's work is supported by Jiangsu 973 project (No. BK2011022), Natural Science Foundation of China (No. 61170035) and NUST project (No. 2011YBXM18).

6 References

[1] Kevin Ashton. “That 'Internet of Things' Thing”; RFID Journal, 22 July 2009.

[2] ITU Internet Reports. “The Internet of Things”; November 2005.

[3] A. Jules. “RFID security and privacy: a research survey”; IEEE Journal on Selected Areas in Communications, vol. 24, Issue 2, pp. 381–394, 2006.

[4] “Wireless sensor network”; http://en.wikipedia.org/wiki/Wireless_sensor_network.

[5] Dargie Waltenegus and Poellabauer Christian. “Fundamentals of Wireless Sensor Networks: Theory and Practice”; Wiley Publishers, Inc., USA, 2010.

[6] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. “Wireless sensor network survey”; Computer Networks, vol. 52, pp. 2292–2330, 2008.

[7] LIU Pu and LI Chao. “Composition Principle of Internet of Things Architecture”; Information Technology, vol. 1-2, pp. 9-12, 2011.

[8] D. Donoho. “Compressed sensing”; IEEE Transactions on Information Theory, vol. 52, Issue 4, pp. 1289- 1306, 2006.

[9] D. Donoho and Y Tsaig. “Extensions of compressed sensing”; Signal Processing, vol. 86, Issue 3, pp. 533- 548, 2006.

[10] E. J. Candès. “Compressive sampling”; in Int. Congress of Mathematicians, Madrid, Spain, vol. 3, pp. 1433–1452, 2006.

[11] E. J. Candès and M. B. Wakin. “An introduction to compressive sampling”; IEEE Mag. Signal Proc., vol. 25, Issue 2, pp. 21–30, Mar 2008.

[12] Peter Pacheco. “An Introduction to Parallel Programming”; Morgan Kaufmann Publishers, Inc., USA, 2011.

[13] Jason Sanders and Edard Kandrot. “CUDA by Example: An Introduction to General-Purpose GPU Programming”; Addison-Wesley Publishers, Inc., USA, 2010.

[14] R. G. Baraniuk. “Compressive sensing”; IEEE Signal ProcessingMag.,vol. 24, Issue 4, pp. 118–120, 124, Jul. 2007.

[15] Michael Elad. “Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing”; Springer Publishers, Inc., UK, 2010.

[16] JIAO Li-cheng, YANG Shu-yuan, LIU Fang, and HOU Biao. “Development and Prospect of Compressive Sensing”; Acta Electronica Sinica, vol. 39, Issue 7, pp.1651-1662, Jul. 2011.

[17] SHI Guang-ming, LIU Dan-hua, GAO Da-hua, LIU Zhe, LIN Jie, and WANG Liang-jun. “Advances in Theory and Application of Compressed Sensing”; Acta Electronica Sinica, vol. 37, Issue 5, pp.1070-1081, Jul. 2009.

[18] G. Peyré. “Best Basis compressed sensing”; Lecture Notes in Computer Science, vol. 4485, pp. 80-91, 2007.

[19] E. Candès, J. Romberg, and T. Tao. “Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information”; IEEE Trans. on Information Theory, vol. 52, Issue 2, pp. 489-509, 2006.

[20] R. Baraniuk. "A lecture on compressive sensing"; IEEE Mag. Signal Processing, vol. 24, Issue 4, pp. 118-121, 2007.

[21] R. Coifman, F. Geshwind, and Y. Meyer. "Noiselets"; Appl. Comp. Harmonic Analysis, vol. 10, pp. 27–44, 2001.

[22] R. G. Baraniuk, M. Davenport, R. DeVore, and M. B. Wakin. "The Johnson-Lindenstrauss lemma meets compressed sensing"; dsp.rice.edu/cs/jlcs-v03.pdf, 2006.

[23] S. S. Chen, D. L. Donoho, and M. A. Saunders. "Atomic decomposition by basis pursuit"; SIAM Review, vol. 43, Issue 1, pp. 129-159, 2001.

[24] FANG Hong and YANG Hai-Rong. "Greedy Algorithms and Compressed Sensing"; Acta Automatica Sinica, vol. 37, Issue 12, pp.1413-1421, December, 2011.

[25] R. Neff and A Zakhor. "Very low bit rate video coding based on matching pursuits"; IEEE Transactions on Circuits and Systems for Video Technology, vol. 7, Issue 1, pp. 158-171, 1997.

[26] J. Tropp and A. Gilbert. "Signal recovery from random measurements via orthogonal matching pursuit"; IEEE Transactions on Information Theory, vol. 53, Issue 12, pp.4655−4666, 2007.

[27] D. L. Donoho, Y. Tsaig and I. Drori. "Sparse solution of underdetermined linear equations by stage wise orthogonal matching pursuit"; Technical Report, pp. 1-39, 2006.

[28] D. Needell and R.Vershynin. "Signal recovery from inaccurate and incomplete measurements via regularized orthogonal matching pursuit"; IEEE Journal of Selected Topics in Signal Processing, vol. 4, Issue 2, pp. 310−316, 2010.

[29] M. A. Figueiredo, R. D. Nowak, and S. J. Wright. "Gradient projection for sparse reconstruction: application to compressed sensing and other inverse problems"; IEEE Journal of Selected Topics in Signal Processing, vol. 1, Issue. 4, pp. 586−597, 2007.

[30] I. Daubechies, M. Defrise, and C. De. Mol. "An iterative thresholding algorithm for linear inverse problems with a sparsity constraint"; Communications on Pure and Applied Mathematics, vol. 57, Issue 11, pp. 1413−1457, 2004.

[31] J. F. Cai, S. Osher, and Z. W. Shen. "Linearized Bregman iterations for compressed sensing"; Mathematics of Computation, vol. 78, Issue 267, pp. 1515−1536, 2008.

[32] A. C. Gilbert, S. Guha, and P. Indyk. "Near-optimal sparse Fourier representations via sampling"; Proceedings of the Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, STOC'02, pp. 152 - 161, 2002.

[33] A. Gilbert, M. Strauss, J. Tropp, and R. Vershynin. "Algorithmic linear dimension reduction in the L1 norm for sparse vectors"; In: Proceedings of the 44th Annual Allerton Conference on Communication, Control and Computing, http: / / www. math. ucdavis. edu/ ~ vershynin/ papers/ algorithmic-dim-reduction.pdf.

[34] G. Gormode and S. Muthukrishan. "Combinatorial algorithms forcompressed sensing"; IEEE, In: Proceedings of the 40th Annual Conference on Information Sciences and Systems. Princeton, USA, pp. 280−294, 2006.

[35] G. S.Almasi, A. Gottlieb. "Highly Parallel Computing"; Benjamin-Cummings publishers Co., Inc. Redwood City, CA, USA, ISBN:0-8053-0177-1, 1989

[36] Zhang Chunmei, Yin Zhongke, Chen Xiangdong, and Xiao Mingxia. "Signal over-complete representation and sparse decomposition based on redundant dictionaries"; Chinese Science Bulletin, vol. 50, Issue 23, pp. 2672-2677, 2005.

Parallel Random Search Algorithm for Large-Scale Constrained Pseudo-Boolean Optimization Problems

L.A. Kazakovtsev
Department of Information Technologies, Siberian State Aerospace University, Krasnoyarsk, Russia

Abstract - *Random search methods are successfully implemented for variety of discrete optimization NP-hard problems when any exact solution approaches cannot be implemented due to large computational demands. Initially designed for unconstrained optimization, the probability changing method gives an approximate solution for various linear and non-linear pseudo-Boolean optimization problems with constraints. Although, in case of large-scale problems, the computational demands are also significant and the precision of the result depends on the spent time. For constrained optimization, the search of any feasible solution may take significant computational resources.*

In this paper, we consider an approach to developing parallel versions of the algorithms based on the modified probability changing method for constrained pseudo-Boolean optimization. Optimization algorithms are adapted for the systems with shared memory (OpenMP) and cluster systems (MPI). The parallel efficiency is estimated for the large-scale non-linear pseudo-Boolean optimization problems with linear constraints and travelling salesman problem.

Keywords: Discrete optimization, parallel computing, MPI

1 Large-scale pseudo-Boolean optimization problem

Most exact solution approaches to the problem of discrete (combinatorial) optimization (knapsack problem, the traveling salesman problem etc.) are based on branch-and-bound method (tree search) [1,2,5]. Unfortunately, most of such problems are in the complexity class NP-hard and require searching a tree of the exponential size and even parallelized versions of such algorithms do not allow us to solve some large-scale pseudo-Boolean optimization problems in acceptable time without significant simplification of the initial problem.

The heuristic random search methods do not guarantee any exact solution but random search methods are statistically optimal. I.e. the percent of the problems solved "almost optimal" grows with the increase of the problem dimension [1].

Let's consider the problem:

$$F(X)=\left(\sum_{i=1}^{N\cdot V} a_i x_i\right)\left(\sum_{i=1}^{N\cdot V} c_i x_i\right)\rightarrow max; \qquad (1)$$

$$\sum_{i=1}^{N\cdot V} b_{ik} x_i \le B_k \;\forall\; 1\le k\le N_{constr} \qquad (2)$$

$$\sum_{i=l\cdot(V-1)+1}^{l\cdot V+1} x_i \le 1 \;\forall\; 1\le l\le N; \; . \qquad (3)$$

Here, $a_i, c_i, b_i \; (1\le i\le N\cdot V)$ are some constants, $x_i \; (1\le i\le N)$ are Boolean variables. The objective function is non-linear.

The real large-scale problems have sometimes millions of variables. For example, the problem of assortment planning of the retail trade company [6] may include thousands goods names to be selected which can be shipped from hundreds suppliers and have 3-10 variants of retail price. In general, problems of such kind can be solved only with random search algorithms.

Being initially designed to solve the unconstrained optimization problems, the probability changing method (MIVER) is a random search method organized by the following common scheme [1,2].

1. k=0, the starting values of the probabilities P_k={p_{k1}, p_{k2}, ... , p_{kN}} are assigned where $p_{kj}=P\{x_j=1\}$. Correct setting of the the starting probabilities is a very significant question for the constrained optimization problems.
2. With probability distributions defined by the vector P_k, we generate a set of the independent random points X_{ki}.
3. The function values in these points are calculated: $F(X_{ki})$.
4. Some function values from the set $F(X_{ki})$ and corresponding points X_{ki} are picked out (for example, point with maximum and minimum values).
5. On the basis of results in item 4, vector P_k is modified.
6. k=k+1, if k<R then go to 2. This stop condition may differ.
7. Otherwise, stop.

To be implemented for the problems like (1-3), this method has to be modified. The modified version of the variant probability method, offered in [6,7] allows us to solve large-scale problems with dimensions up to millions of Boolean variables.

In case of the large-scale problems, even the calculation of the linear objective function takes significant computational resources. The number of constrains usually also grows with the increase of the problem dimension. So, the calculation of the objective function and the constraints is a very large computational problem if it is repeated lot of times.

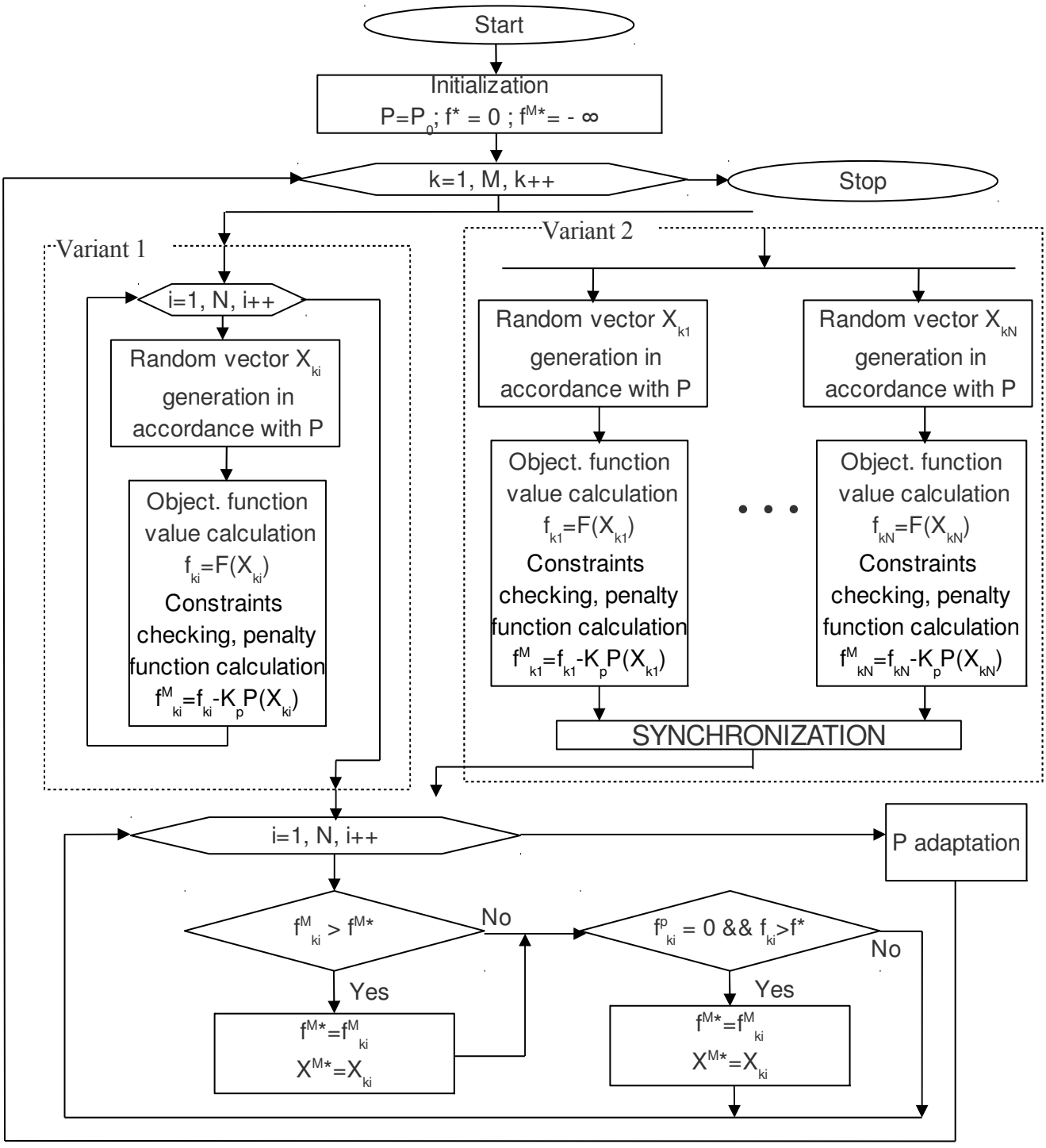


Figure 1. Parallel algorithm for systems with shared memory

That is why, the distribution of the computational tasks between the parallel processors or cluster nodes is very important.

In this paper, we do not consider greedy search algorithms (which are deterministic or also randomized [8, 10]) though they are often used to improve the results of the random search methods as the final step of them.

Also, we do not consider the parallel genetic algorithms [11] though some approaches offered for genetic algorithms may be implemented for random search algorithms parallelization.

Here, we offer an approach of adaptation of the existing programs realizing the random search methods of constrained pseudo-Boolean optimization to be implemented in the parallel systems.

2 Serial version of the algorithm and its parallelization with OpenMP

The scheme of the algorithm for the serial systems is shown in Figure 1, variant 1.

At the step of initialization, all the variables p (components of the probability vector P) are set to their initial values ($0<p_i<1 \quad \forall \quad p_i \in P$). Since our algorithm realizes the method of constrained optimization, the initial value of the probability variables is sometimes very important. Then, we generate the vectors X of optimized boolean variables. In our case of constrained problem of knapsack type, the large values of vector P components generate the values of X which are out of the feasible solutions area due to the constraints (9,10). Due to the constraints (3), the optimal initial values of the vector P components do not exceed $1/(L+1)$ [7].

We set the initial values of the vector P to $1/(L+1)$ but we have to reduce this value if several starts of our algorithm give us no results in feasible solutions area. This process is not illustrated in the Figure 1 for the simplicity.

Instead of maximum number of steps (M in Figure 1), we can use the maximum run time as the stop condition. In some cases, it is reasonable to use the maximum number of steps which do not give us the result better than previous ones as the stop condition.

In the cycle (i=1,N), we generate the set of N vectors X_{ki} in accordance with the probability vector P. Then, the objective function is calculated for each X_{ki}. Also, we calculate the values of the left parts of our constraints, and introduce the second objective function f^p (penalty function):

$$f_{ki} = F_P(X) = C_{PENALTY} \sum_{k=1}^{N_{CONSTR}} F_{Pk}(X); \quad (4)$$

$$f_{ki}^P = F_{Pk}(X) = \begin{cases} 0, \sum_{i=1}^{N \cdot V} b_{ik} \le B_k; \\ \dfrac{\sum_{i=1}^{N \cdot V} b_{ik}}{B_k}, \sum_{i=1}^{N \cdot V} b_{ik} > B_k. \end{cases} \quad (5)$$

Here, $C_{PENALTY}$ is some coefficient. If some estimation of the maximum value of the objective function is available, we can use it as the value of the coefficient $C_{PENALTY}$. For linear objective functions:

$$C_{PENALTY} = \sum_{i=1}^{N \cdot V} |a_i| . \quad (6)$$

The modified objective function f^M is the sum of the objective function and penalty.

When all the values of f_{ki} and f^M_{ki} are calculated, we choose the best (maximum) value for f^M_{ki} and, if there were the variants of the vector X_{ki} which satisfy all the constraints (f^p_{ki}=0) then we choose the maximum value of the objective function f_{ki} for that variants of the vector X_{ki}.

In case of the constrained optimization, we have to use a special method of adaptation of the vector P. The solution of different practical tasks shows the best result if we use the multiplicative adaptation with rollback procedure [2,6]. In this case, the components of the vector P are never set to the value of 0 or 1 which may cause that all the further generations of the X vector have the same value of the corresponding component which does not give the feasible solution due to constrains (3).

$$p_{k,j} = \begin{cases} p_{(k-1),j} \cdot d, & x_{kj}^{max} = 1 \wedge x_{kj}^{min} = 0 \wedge p_{(k-1),j} < 0.5, \\ 1 - \dfrac{1 - p_{(k-1),j}}{d}, & x_{kj}^{max} = 1 \wedge x_{kj}^{min} = 0 \wedge p_{(k-1),j} \ge 0.5, \\ \dfrac{p_{(k-1),j}}{d}, & x_{kj}^{max} = 0 \wedge x_{kj}^{min} = 1 \wedge p_{(k-1),j} < 0.5, \\ 1 - \left(1 - p_{(k-1),j}\right) \cdot d, & x_{kj}^{max} = 0 \wedge x_{kj}^{min} = 1 \wedge p_{(k-1),j} \ge 0.5, \\ p_{(k-1),j}, & x_{kj}^{max} = x_{kj}^{min}. \end{cases} \quad (7)$$

Here, d is the adaptation coefficient. In case of multiplicative adaptation, it does not depend on the step number k. In this case, the absolute value of adaptation step depends on the corresponding value of p_{kj}.

After several steps, the values of P vector elements are close to 0 or 1 and the decrease of the adaptation step (d) results in generation of the similar vector X exemplars which correspond to some local maximum. The rollback procedure is helpful to avoid that situation. It sets the values of P vector and its adaptation step d_k to initial (or other) values. In simplest case, rollback is performed after several steps which do not improve the best objective function value.

The best results are demonstrated with methods of partial rollback procedure which change some part of P vector components or change all the components so that their new values depend on previous results. We can use the following rollback formula:

$$p_{kj} = (p_{k-1\,j} + q_k p_0)/(1+q_k), \text{ if } p_{k-1\,j} < p_0. \quad (8)$$

Here, p_0 is the initial value of the probability. The coefficient q_k may be constant or vary depending on the results of previous steps. For example, it may depend on the quantity of the steps which do not improve the maximum result (s_m).

$$q_k = w / s_m. \quad (9)$$

The weight coefficient w has to be chosen experimentally.

In case of constrained optimization, the choice of the initial value p_0 can be very important. In some cases, the incorrect initial value causes the generation of X vector sets that lay out of the feasible solutions area. After several steps, the penalty function minimization process results in adaptation of P vector which allows to generate X vector exemplars that satisfy our constraint conditions. But the rollback procedure returns the probability vector to its initial (usually incorrect) value. Let's consider a simplest example. Let our problem have only one constraint like that:

$$b_1x_1 + b_2x_2 + b_3x_3 + \ldots + b_Dx_D < B. \quad (10)$$

Here, D is the dimension of our problem (D=N V).

The elements of the vector X at the first steps are generated so that it is set to the value of 1 with probability p_0. The expectation of the left part of (10) is

$$M = p_0 b_1 + p_0 b_2 + p_0 b_3 + \ldots + p_0 b_D = (b_1 + b_2 + b_3 + \ldots + b_D)\, p_0. \quad (11)$$

The maximum values of the objective function are usually achieved at the points where the condition is barely satisfied ($b_1x_1+b_2x_2+b_3x_3 + \ldots + b_Dx_D \sim= B$ in our case). Therefore, the optimal value of the initial probability is

$$p_0 = B/(b_1+b_2+b_3 + \ldots + b_D). \quad (12)$$

The negative effect of the rollback procedure can be reduced by adaptation of the initial value of p_0:

$$p_{0k} = (p_{1\,k-1}+p_{2\,k-1}+\ldots+p_{D\,k-1})/D. \quad (13)$$

This version of rollback procedure is most actual in case of partial rollback which performed at each step of our algorithm. The implementation of this kind of adaptation&rollback procedure is described below for the parallel version of the optimization algorithm for systems with no-remote memory access (clusters).

The adaptation of our algorithm for multiprocessor systems with shared memory can be performed by the parallel generation of the exemplars of the X vector and their estimation. The scheme of that version of our algorithm is shown in Figure 1, variant 2. If our system has N_P processors, the cycle of generation of N exemplars of the vector X can be divided between the processors. Each processor has to generate N/Np exemplars of the vector X and calculate the value of the objective function, left parts of the constraint conditions and calculate the modified objective function values.

Organizing of the parallel thread takes significant computational expenses. In [14], they estimate that expenses as 1000 operations of real number division. The experiments at 4-processor system with linear 100-dimension problem (105 constraints) show that the parallel version runs 2.8 times faster than the serial one.

3 Version for clusters

Our experiments [6,7] with adaptation of random search discrete optimization algorithms [1,2] for parallel execution gave us rather effective results with use of multiprocessor systems with shared memory and OpenMP [14]. But the intensive data interchange in running mechanisms of MPI reduce the efficiency of the realization of the same algorithms for PVM and MPI clusters when the number of the nodes increases. That issue makes the usage of the computing cluster ineffective even in comparison with low-cost multi-core systems.

For testing purposes, we used the following form of the objective function and constraints:

$$F(X) = \left(\sum_{i=1}^{N\cdot V}\sum_{j=1}^{V} a_{ij}x_{ij}\right)\sum_{i=1}^{N}\left(1 - c_i x \sum_{j=1}^{V} x_{ij}\right) \to max; \quad (14)$$

with constraints:

$$\begin{cases} \sum_{i=1}^{N}\sum_{j=1}^{V} b_{ij1}x_{ij} \le B_1; \\ \sum_{i=1}^{N}\sum_{j=1}^{V} b_{ij2}x_{ij} \le B_2; \\ \ldots \\ \sum_{i=1}^{N}\sum_{j=1}^{V} b_{ijN_{Constr}}x_{ij} \le B_{N_{Constr}}; \end{cases} \quad (15)$$

$$\sum_{j=1}^{V} x_{ij} \le 1 \;\forall\; 1 \le i \le N. \quad (16)$$

The coefficients in the constrains in the test set are generated so that the feasible solutions set is 1000-1000000 smaller than 2^N (depends on N and randomly selected values). The algorithm replaces the whole set of constraints with penalty function [2, 6, 7].Problem of a traveling salesman [1, 2, 11]:

$$F(X) = -\sum_{i=1}^{N} a_{ij}x_{ij} \to max; \quad (17)$$

The only constraint in this case is that the matrix of Boolean variables X must describe the adjacency matrix of Hamiltonian graph. This cycle must be the only path in that graph. Here, a_{ij} is the distance (cost, time, etc.) between i-th and j-th places (cities), $a_{ii} = 0\ \forall\ i$, $aij = a_{ji}\ \forall\ i,j$.If $x_{ij}=1$ then it means that the salesman must go from i-th place to the j-th one.

We tried to solve the optimization problems in Argo cluster (SISSA, Trieste, Italy). The program code was adapted for MPI running. Algorithm designed for OpenMP parallelizing (see Figure1, variant 2) was implemented for paralellizing via MPI. MPI synchronizing procedure was used at the step "Synchronization". Following this flowchart gave negative results in comparison even with a single node (8 nodes, 2 processes per node, 50 variants of X sets in each generation on each node, 10000 Boolean variables, 11000 constraints).

Analysis of the problem showed that at the step of synchronization, each node spent approximately 12% of time waiting for the other nodes to finish the calculation, data interchange between nodes in this case is rather intensive (10000 Boolean variables had to be sent from each node, then, each one had to receive 10000 real numbers of the probability vector after each step).

Reconstruction of the algorithm so that each node performs N steps (100-500 for that scale of the problem) separately, and comparing the results after the N-th step, choosing the best ones by master nodes and broadcasting new probability vector gave much better results. But, in this case, each node had to spend up to 43% of time (!) waiting for the others to complete their calculations.

Another way to organize the parallel execution of the search processes with the maximum independence from each other is to organize multiple starts of the algorithm at all the nodes. The algorithm starts at the nodes with the same or different initial parameters (for example, different initial values of the probability vector elements).These

multiple simultaneous starts are performed instead of the rollback procedure in the serial version of the optimization algorithm. Each node starts the cycle of the random X vector generation with probability vector adaptation. If the algorithm does not improve the best objective function value during several steps, the rollback procedure is performed. When all the nodes reach the stop condition, the results of all the nodes are compared to figure out the best one as the final result of the optimization problem.

In its simplest form, the above approach does not need any modification of the implemented software. The serial version of the algorithm runs at all the nodes simultaneously and the results of the nodes are then compared by the operator or by the special node.

The simultaneous execution of the serial version of the algorithm improves the results of the calculations insignificantly due to the similar behavior of the algorithm at all the nodes. The comparison of the P vector values from different nodes after several steps shows that the difference between them is reduced with each following step and the nodes generate the new exemplars of the X vectors in the very close area. In case of the algorithm with no average probability adaptation, the simultaneous execution of the serial version of the algorithm with different initial probability value gives the results much better than the serial algorithm executed at the single node. But in most cases, the version of the optimization algorithm with the average probability adaptation executed at the single node shows even better results. So, our approach must support the average probability adaptation for the optimization algorithms executed at different nodes (13).

a) Initialization (at each node), **MPI_INIT**
b) Generating N random sets of variables X, estimating F(X) and constraints (at each node)
c) Choosing the best and the worst X sets, adaptation of the probability vector P (at each node separately)
d) if I am the master node then:
d1) check if any data transfer started at the previous steps has been completed
d1.1) if so, then if it was the maximum result reached by the k-th node, check if this result is better than the global result
d1.1.1) if so, send message "OK" to the k-th node
d1.1.2) if not so, send "NOT OK" to the k-th node and send the actual global maximum and the corresponding X vector to the k-th vector
d1.2) if the completed data transfer contains the best vector X
d1.2.1) then recaclulate F(X) and if it is better than the global maximum, assume that this is the new global maximum
d2) if there are any messages from the other nodes waiting to be received and no receiving processes are in progress, start receiving (non-blocking, continue processing)
d3) check if the stop conditions are reached. **If so, send "STOP" message to all the processes, wait for all of them to receive it and then MPI_FINALIZE, stop**
e) if I am not the master node and I have already started any sending process then
e1) check if this process has been completed
e2) if so, start receiving of the answer from the master node
f) if I am not the master node and I am waiting for a response from the master then
f1) check if the master node has started sending anything
f2) if so, start receiving (non-blocking)
g) if I am not the master node and I have already started receiving smth. from the master node then
g1) if the receiving has been completed then
g1.1) if it was "OK" message, start sending my best X vector (non-blocking)
g1.2) if it was "NOT OK" message, start receiving global best X vector (non blocking)
g1.3) if it was the global best X vector, recalculate F(X) and, if it is better than my own maximum, partially restart local process to perform further search around the global maximum
g1.4) if it was the "STOP" message, then stop, MPI_FINALIZE
h) Checking for the restart conditions (at each node separately), algorithm has to be restarted if it hasn't improved the best results during several last steps
i) if the restart conditions are reached then
e1) if I am not the master node, start sending the best F(X) value
e2) reset probability vector
j) go to step b

Figure 2. Parallel algorithm description

The algorithm was re-written (Figure 2) so that all the data transfer was performed at the step of checking of the restart conditions (text in bold used for the steps performing message passing). If the algorithm does not improve the local maximum values of the objective functions after c_{max} generations, the process tries to communicate to the other processes to check if any other node has improved the previous global maximum. If so, the process receives that value and the corresponding X vector values. If the local maximum value is better than the received one then we report our values as the new global maximums.

In our algorithm, we do not send P vector values which are rather long. The vector P for the problem with 10000 variables needs 40000 bytes to be sent. The node having achieved global optimum sends this value and the values of the vector X_G^* (1250 bytes in case of 10000 Boolean variables). The values of the new probability vector are calculated as

$$p_i = \frac{(C_{corr} + (1 - 2C_{corr}) x^*_{Gi}) p_G^{AVG}}{C_{corr} + (1 - 2C_{corr}) p_G^{AVG}} \quad . \qquad (18)$$

Here, $x^*_{G\,i}$ is the i-th element of the received vector X_G^*, D is the dimension of the problem (number of the elements of X_G^*), C_{corr} is some small real value, p_G^{AVG} is the average value of the probability vector generated the vector X_G^*. If the received message does not contain it then the node which has received the message evaluates it:

$$p_G^{AVG} \approx \frac{\sum_{i=1}^{D} x^*_{Gi}}{D} \quad . \qquad (19)$$

Taking into consideration the constraints (3) if they exist, the value of the constant C_{corr} can be calculated as $0.5/V$.

The last version of the algorithm, though it is rather efficient, has very low data interchange between nodes (log shows that there was only 31-152 data transfer sessions for each node during 1 hour). So, with this intensity of data traffic, there is no need to implement any expensive network to build an efficient MPI cluster for random search problems solution.

4 Results

In case of the problem (14), the results achieved by the program working on 8 nodes separately with comparison of the best result after the last step (4 nodes, 2 processes per node, 10000 Boolean variables, 11000 constraints, non-linear objective function) this version of the algorithm achieves in half time (2 hours for the separately working nodes and 63 minutes for this version of the algorithm).

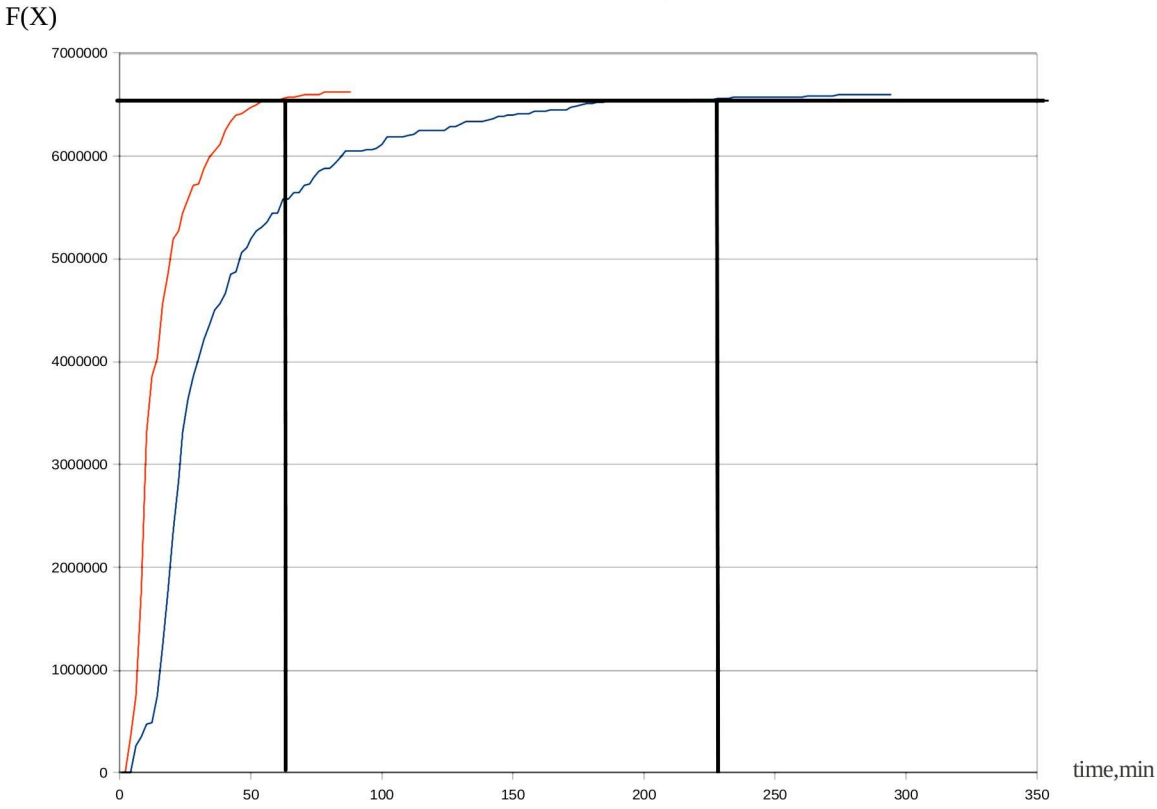


Figure 3. Comparison of the results of serial and parallel versions of the algorithm

Traveling salesman problem with 1000 points (1000000 Boolean variables) after 4 hours of calculation on 4 nodes had not given the exact results, the results were still improving at the last steps, problem with 500 points (250000 variables) after 4 hours of calculation gave rather good result, very close to the optimal solution.

The parallel efficiency for the cluster system of 6 nodes is 0.81. An example (Figure 3) shows the results of

running of the algorithm on cluster and on a single computer which is a part of that cluster. To build this diagram, we included in our algorithm special block which stores the maximum objective function value reached by the algorithm and the current time after each 10 steps to a special array. The horizontal line shows when the algorithm achieves the control value (65015.17). To reach this control value, the serial version of the algorithm has spent 229 minutes (13748 seconds), the parallel version has spent 59 minutes (3556 seconds).

The average value of the parallel efficiency (0.81) is calculated as the average speed-up coefficient after 10 runs for 5 different objective functions. It is interesting, that at the very first steps, the parallel efficiency coefficient is less than at the following steps. The possible reason of this fact is the process of parameters tuning which is performed by each node at the first steps and the intensive message passing.

5 Conclusions

- Blocking MPI operations give negative results in comparison with non-blocking ones for the random search problems. In case of fine granularity parallelization [12, 13], results are negative even in comparison with a single node;

- Though the realization of the simplest protocol with 6 kinds of messages is rather a complicated problem, non-blocking operations work fine, their implementation gives significant (2x) speedup in comparison with the simultaneous running of the same code at separate nodes;

- Using the high-performance clusters increases the maximum scale of the problems solved;

- To build the MPI cluster for random search problems, expensive high-performance network is not needed because of very low intensity of data traffic;

- Implementation of MPI does not decrease the productivity of a single node, OpenMP still gives perfect results in comparison with serial code.

6 References

[1] Alexander N. Antamoshkin. “Optimization of Functionals with Boolean Variables”, Tomsk University Press, Tomsk, 1987 (*in Russian*)

[2] Alexander N. Antamoshkin (edited by) “Systems Analysis: Design, Optimization and Application”, Siberian Airspace Academy, Krasnoyarsk, 1996

[3] N. Baba “Convergence of Random Search Optimization Methods for For Constrained Optimization Methods”; Journal of Optimization Theory and Applications (Springer), issue 33, pages 451-461, Apr. 1981

[4] John R.Birge, Charles H.Rosa “Parallel Decomposition of Large-Scale Stochastic Nonlinear Programs”; Annals of Operation Research (Baltzer AG), issue 64, pages 39-65, 1996

[5] Yuri Finkelstein et all, “Discrete Optimization”; in: Optimization Methods in Economical-Mathematical Modelling, Nauka, Moscow, pages 350-367, 1991 (*in Russian*)

[6] Lev A. Kazakovtsev “Adaptation of the Variant Probability Method for the Problems of Assortment Planning and Others”; Vestnik NII SUVPT (Research Institute of Control Systems, Wave Processes and Technologies, Krasnoyarsk), issue 6, pages 42-50, 2001

[7] Lev Kazakovtsev “Adaptive Model of Static Routing for Traffic Balance between Multiple Telecommunication Channels to Different ISPs”, http://eprints.ictp.it/415

[8] P.M. Pardalos, L. Pitsoulis, T. Mavridou, M. G. C. Resende “Parallel Search for Combinatorial Optimization: Genetic Algorithms, Simulated Annealing, Tabu Search and GRASP”; Parallel Algorithms for Irregularly Structured Problems: Lecture Notes in Computer Science, issue 980, pages 317- 331, 1995

[9] G. Rudolph "Parallel Approaches to Stochastic Global Optimization"; Parallel Computing: From Theory to Sound Practice, Proceedings of the European Workshop on Parallel Computing EWPC 92 (IOS Press, Amsterdam), pages 256-267, 1992

[10] H. Mühlenbein, M. Shomisch, J. Born “The parallel Genetic Algorithm as Function Optimizer”; Proceedings of the fourth Conference of Genetic Algorithms (Morgan Kaufmann, San Mateo), pages 271-278, 1991

[11] M.G.C. Resende, C.C. Ribero "Greedy Randomized Adaptive Search Procedures"; International Series in Operations Research & Management Science, Vol. 57 "Handbook on Metaheuristics", pages 219-249, 2003

[12] Salvatore Rinaudoy, Francessco Moschellay, Marcello A. Anilez “Controlled Random Search Parallel Algorithm for Global Optimization with Distributed Processes on Multivendor CPUs”; http://citeseer.ist.psu.edu/710446.html .

[13] Piet Spiessens, Bernard Manderick “A Massively Parallel Genetic Algorithm: Implementation and First Analysis”; Proceedings of the fourth Conference of Genetic Algorithms (Morgan Kaufmann, San Mateo), pages 279-286, 1991

[14] Leonardo Dagum, Ramesh Menon “OpenMP: An industry-standard API for shared-memory programming”, IEEE Computational Science and Engineering (IEEE Computer Society Press), Vol. 5, Issue 1, pages46–55, Jan.–Mar. 1998

The discussion of energy conservation of data center from the evaporative cooling technology of HPC

Ruan, Lin*, Li Zhenguo
Institute of Electrical Engineering, Chinese Academy of Sciences, Beijing P.R.China 100190

Abstract - *The IT industry is developing fast and more and more large scale data center and supercomputing center were built or in building, which improve scientific research and data service also bring about huge energy consumption. This paper aims to make a discussion of energy conservation of data center from the evaporative cooling technology of HPC. Firstly, the developing tendency of data center and the traditional cooling technology are introduced; secondly, much more concentration were put on the newly high performance evaporative cooling technology for HPC and we showed a prototype of the spray evaporative cooling HPC developed by IEECAS, finally, a prospective blueprint was drawn up for the future construction of data center with evaporative cooling HPC.*

Keywords: Energy conservation; Evaporative cooling technology; Data center; HPC

1. Perface

During more than one hundred years since the industrial revolution, the people's demand for the energy is dramatically increasing, which gives much more pressure to the supply of the energy and also bring about the serious environmental crisis.

With the deepening of the informationization development, the super computer or high performance computer (HPC) is developed quickly and being extensively applied in many fields.

Data center provided with lots of IT equipments especially HPC makes great contribution to the scientific researches especially in the field like astronomy and meteorology relating to a lot of data processing and computing.

Whereas, with the continuously improvement of the data processing capability, computing speed and storing ability, the data center will consume a huge amount of electricity during its daily operation and at the same time brings about serious energy consumption problem to the society.

Besides the energy consumption of the hardware in the data center, in order to secure the normal and safety operation of the IT equipment, an additional electrical energy must be used to dissipate the heat. The cooling problem of the IT equipment shows up and becomes a big challenge for the cooling technology. Among that, the energy consumption of HPC is predominant.

Aiming at the HPC cooling, the traditional method include air cooling, water cooling and heat pipe cooling etc. Water cooling and heat pipe are belong to closed loop cooling method, the manufacture technology is a little bit complex and the cost is relatively higher than others. The most important is the water cooling has the potential of leakage which will lead to an electrical accident. Air cooling is easiest and has been widely used in HPC. But the disadvantage is that is the noise is big and the cooling capability is limited. Due to its usage of air conditioner, the energy cost is very high. According to the statistic, in a nowadays air cooling data center , the minimum of 50% average carbon exhaust come from the cooling system rather that the IT equipment itself.

Recently, the research of high effective cooling technology is becoming the hot point globally and promoting the energy efficiency is a huge challenge. Evaporative cooling technology takes advantage of phase change to realize the cooling effect of the object. Its

cooling capability is far larger than the other heat transfer method using specific heat.

This cooling technology has been successfully used to the large electrical machine in china and we have priority of intellectual property right. The recent application of this technology to the large electrical machine is the world famous Three Gorges hydropower project. It's the largest hydropower plant in the world. There are two sets of 840MVA hydrogenerators using this kind of cooling technology designed by IEECAS.

The characteristic of this cooling technology include: high efficiency for the heat transfer, high safety and most important low energy cost or zero energy cost from the cooling system itself. All of these characteristics show cases it will be very suitable for the HPC and data center.

2. The development and challenge of the modern data center

Pushed forward by the explosive increase of demand for the data services and rapidly development of IT technology, the data center has experienced a tremendous development and the increase tendency will be lasting, the annual growth rate will be no less than 10%. Whereas, the energy cost of data center becomes un-neglecting.

With the increase of demand for the computing capability, more IT equipment and more density of the equipment in the data center will be needed. The general tendency of the development of data center is limited space with as many as cabinet, high integration level, small sized chips and high density, which means more energy consumption per unit space.

According to statistics, the electricity consumption of data center is increasing annually at the rate of 15-20%, the price for the energy is soaring at the same time. So the maintaining cost for a data center is very high. It was said that the biggest single cost for Google is for the electricity. That is why they want to arrange their data center near a hydropower plant. .

The energy consumption of data center has its specialties, the verity of energy consuming IT equipment is much and the construction is very complicated. In general, the energy consumption of data center mainly covers the following aspects: high performance IT equipment itself (including CPU, RAMS, CHIPSET and external appliances), power supply equipment (power losses during the conversion of power sources) and cooling system. The typical power flow is introduced in the following Fig.1.

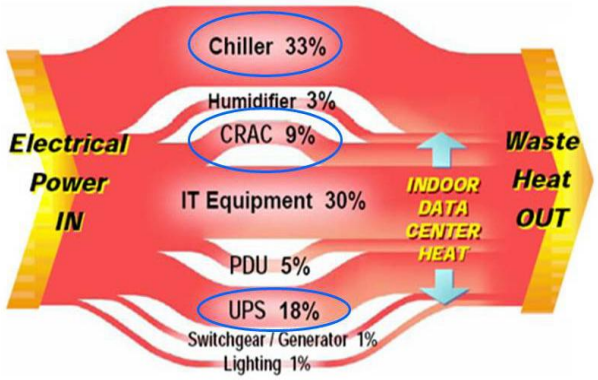


Fig.1 Power flow in a typical data center

Among that, cooling system for data center should not be belittled and it has very close relation with the cooling technology and cooling structure.

The recently built data center and supercomputing center has more concern about the energy saving, but it's still in a high level. Take ShangHai supercomputing center for example, since the Dawning "4000A" super computer system put into operation in August 2004, its stable load rate is relatively high, and the main CPU usage is retaining about 85%, it has a lot of consideration about the arrangement of the rack shelf and the ventilating passage design. Analysis on the basis of statistic of mean quarter operating data of whole supercomputing system showed the energy consumption in the following table1. From this table, we can see that the energy consumption of the cooling system account for more than 50% of that of IT equipment itself. See from this point of view, focusing on the energy saving of the cooling system will be obviously productive for decreasing the energy consumption of data center.

Table1. Energy consumption of "4000A"

Cooling method	Real cases	IT hardware electricity consumption	Cooling system account for

Air cooling	"4000A"	285kW	54.4%

3. The traditional cooling method for the HPC

Recently, the cooling technology is becoming newly researching hot point of HPC, improving the energy efficiency of the computing system and data center is a huge challenge. There are many cooling method for tackle with the heat dissipation of HPC, but from the point of view of putting into commercial use, the most traditional or easiest realizing method is air cooling and water cooling has the potential to show up due to it high performance of heat transfer

3.1 Air cooling

Air cooling is the most-easily-realized cooling method, but it needs the use of CRWC and its cooling effect rely more on the surrounding temperature and its cooling capacity has utmost. Theoretically, the design value for the cooling capacity is 1500w/m2 that means the unit computer cabinet is 6kW, the utmost is 8kW.

The most serious problem relating to the air cooling method is the fan noise and the huge energy consumption. The cooling effect and the noise of the fan is a big contradiction seems cannot be easily solved. This cooling method conform to the convective heat transfer, in order to promote the cooling effect, we must increase the speed of wind flow, so the increase of numbers of fans and the rotating speed of fans is inevitable (normally, the speed for the CPU is above 5000rpm), which lead to big noisy pollution. If for an open air cooling atmosphere, the air flow also arouse the dust flow inside the mainboard and deposited on that, bring about the decrease of safety and maintenance inconvenience.

For air cooling data center, the arrangement of the cabinets and the design for the ventilation passage is very complex, even the design is theoretically practical, and in real application, the operator of data center found that the cooling effect of different cabinet differs a lot from each other, there will appear local high temperature area and in some extreme circumstance, a few cabinet cannot working due to high temperature.

In reality, in air cooling data center and supercomputing center, up to 50% energy cost is not for computation, but for the necessary cooling system, if we concern more about the energy efficiency of the data center, this cooling method is far beyond our desire. And such problem will become more serious with the development of the chip capacity and IT cabinet integration. Lot of giant company such as IBM is struggling to solve this energy problem caused by cooling demand.

3.2 Water cooling

Due to the thermal load limit of the traditional air cooling and some other problem, the alternative is necessary. So, for large processor or supercomputing system, the water cooling method shows up the practical significant. Water is a good fluid having bigger specific heat than air, so the cooling effect can be promoted a lot. Google is programming its largest super computer with two 4-layer building used for cooling system. K-computer with water cooling system (Fig.2 and Fig.3) ranked No 1 in the last year's world top 500 in the supercomputer field.

For cooling method itself, there is no technique knowhow and easy to be extended. But higher safety risk (water is of electrical conductivity and water leakage cannot be easily avoided due to high flowing pressure), higher manufacturing cost (technology is complex and hard to assembly and maintain) and having no positive contribution to the energy saving make it commercial prospective not very well.

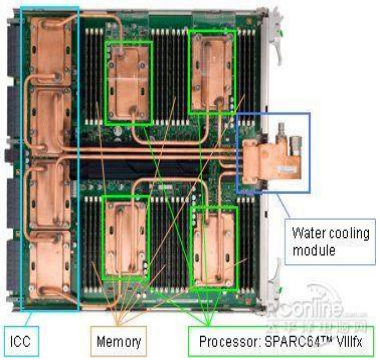


Fig.2 Cooling main board of K computer

Fig.3 K computer cabinet

4. The application of the evaporative Cooling technology to the HPC

The evaporative cooling takes advantage of the physical process of phase change to take away heat losses when an appropriate liquid-phase coolant becomes gas-phase coolant. The evaporative cooling technology is kinds of liquid cooling method combined with specific cooling structure design which can avoid the fatal disadvantages of the water cooling completely due to it adopt some special coolant with high insulation.

The Institute of Electrical Engineering, Chinese Academy of Sciences (IEECAS) has been doing the evaporative cooling research since 1958. During more than 50 years' developing process, the evaporative cooling technology has experienced some stage of innovation and has been used to various electrical equipments, such as turbogenerators, hydrogenerators, electrical power transformers and power electronic component equipment etc.

Due to the urgent demand of high efficiency cooling technology for HPC, we try to use it to the HPC on the basis of our experiences on applying of it to the power electronic component equipment. Its application to the HPC not only can solve the heat dissipation problem but can obtain the goal of energy saving.

Phase change heat transfer is of high efficiency, it can realize the good cooling effect for the HPC, at the same time the stability and safety of HPC can be guaranteed due to the selection of the evaporative coolant. The evaporative coolants are all liquid material with proper boiling temperature best matched with the cooling target. It is primarily environmental protected, is of high insulation, it has on toxicity, no liable to inflame and good chemical and physical stability.

As to energy conservation, for different cooling structure, the energy saving level varies. The following will introduce two kinds of evaporative cooling HPC design. The energy saving target is different.

4.1 Spray evaporative cooling HPC

We have developed a kind of spray evaporative cooling super computer with the cooling medium directly contact with the heated chips, during the phase-changing procedure, the large amount of latent heat will be absorbed by the coolant from the heated chip without increase its temperature, so it very fit for the high density computer cabinet cooling, the cooling effect is better than air cooling at the same thermal load level, and most important effect is that the coolant circulation just need a little momentum provided by the pump, the energy consumption for this kind of cooing system is just for the energy cost of pump.

When the HPC start working, the liquid phase evaporative coolant sprayed out from the nozzle and have a direct contact with the heated chips distributed in the HPC cabinets, part of them vaporizes when absorbing heat and gas phase coolant rise to the condenser. The pump provides the momentum for the coolant circulating in the closed loop. In the equipment, the nozzle is located near the mainboard and spays liquid coolant toward the main heated chips shown in Fig.4.

We have built a prototype of spray evaporative cooling super computer in our lab (Fig.5), it unit cabinet thermal capacity is 50kW, but the max energy cost for the pump is just 3.06 kW accounting for only 6.12%. If a group of computer cabinets operate parallel, the proportion of pump energy cost will decrease due to the power of pump is non-linearly increased. So, for a data center with many hundreds of cabinet, the energy saving effect is more in evidence.

4.2 Self-circulating evaporative cooling HPC

For more energy saving cooling system, we come up with the self-circulating evaporative cooling system for

HPC, which have some thin liquid boxes attach to the heated chips and all these boxes can be connect together or form circulation branches separately.

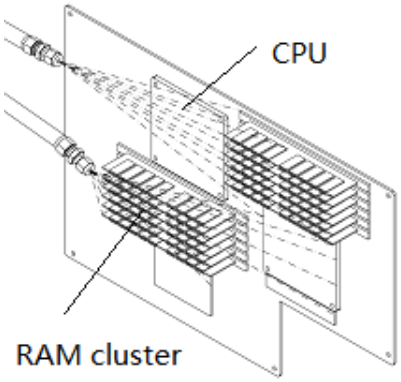


Fig. 4 Spry of coolant to the heated chips

Fig.5 Prototype of spray evaporative cooling HPC

The self-circulating evaporative cooling HPC consists of blade units, blade cases, cabinet, condenser, liquid boxes and some connecting tubes. The principle diagram is shown in the fig.6. the liquid box shown in Fig.7 is fill with the evaporative coolant with the boiling temperature of 40-50℃. When the chip heated, it will transfer the heat to the box attached on it, the coolant inside the box absorb the heat, part of them boiling and liquid coolant becomes gas phase coolant and flow upward along the gas tube due to the density difference and flow back along the liquid tube after it is cooled into liquid phase in the condenser. So, the whole circulation system is closed loop, and the coolant flow momentum is aroused from the density difference between liquid phase and gas phase coolant, there is no need of external forcing momentum. So, there is no energy consumption at all.

The use of thin box brings about additional contact thermal resistance and a thermal conductive link along the direction of box wall thickness. For improve it, some researches on surface conductive and convective enhancement methods are underway in our lab.

In a long run, this kind of cooling system for HPC is the best choice for the data center and supercomputing center. The prototype is under construction in our lab. The cooling system itself can completely realize zero energy consumption and is self-circulating, no fan or pump noise, self-adaptive and safety. The only problem is a little higher primary investment for the fabrication of liquid box due to its odd-shaped. When it can be standardization and realize mass production, the initial investment will not be a problem anymore.

5. The prospect for the construction of the future low energy consumption data center

For the next generation of HPC, air cooling cannot be satisfying and water cooling was hampered by the high cost, safety and maintenance. So, introducing the high efficiency evaporative cooling technology to the IT equipment, especially the data center with centralized use of IT equipment is a good resolution for high cooling effect and lower energy consumption.

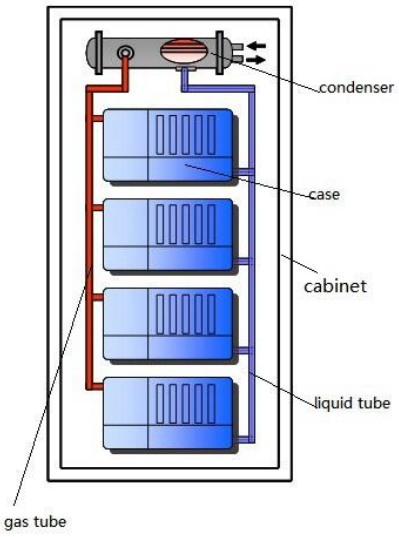


Fig.6 The diagram of self-circulating evaporative cooling HPC

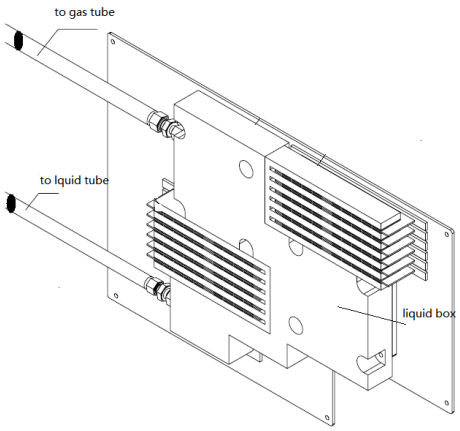


Fig.7 Liquid box with connecting tube

The evaporative cooling system is closed loop and has direct cooling effect to the heated chip, so there is no need of installation of CRWC, it can eliminate to the maximum the noisy pollution aroused form the fan of air cooling in the data center and can save a lot of money for the electricity usage relating to the cooling system. It can provide a clean, high capacity, quiet environment for the future scientific research and large scale numerical calculation. By adopt micro-force circulation and self-circulation system to realize the high-performance of cooling system, it provide more rooms for increase the density of single chip; it can also increase the density of cabinet or data center; it has more flexibility to get a good use of space of data center. Due to its low or zero energy cost, it will have a great contribution to push forward the energy conservation of data center and with no doubt will play a positive influence on the sustainable development of IT industry.

Furthermore, lower power consumption or zero power consumption evaporative cooling technology combined with the architecture energy-saving and the reasonable usage of afterheat should attract more attention when sketching the future construction of data center and some auxiliary facilities.

6. References

1) Yinqi, zhang,etc. “The analysis of the promotion of energy efficiency of HPC system in ShangHai super computer center”, computer world, version 22, 2010.

2) DaWei,Gu,etc. The instruction and construction of data center, Publishing House of Electronics Industry, 2010

3) Lin Ruan, etc. “Surface liquid box attachment evaporative cooling equipment for super computer” 200910243551.X.

4) Lijing,Gu,etc. “The research on the energy consumption and energy efficiency of data center in our country”, china energy, Vol. 32 No. 11 Nov 2010

5) Feng Zhao, “the introduction of energy conservation for data center”, telecommunication network technology, vol.1.2011.pp: 5-8

6) Zhang, Peng ; Ruan, Lin,etc. “Design of Experimental Device Used for Heat Transfer Researchof Single-phase and Two-phaseSprayCooling”, ICEMS,2011.pp: 1 – 3

7) Peng Zhang, Lin Ruan* “experimental study on two-phase spray cooling for the cooling of high-heat-flux electronic chip”, CDCIEM2012, pp: 324 - 326.

SESSION

CLUSTER COMPUTING + MULTI-CORE, GPU, FPGA PROCESSING AND APPLICATIONS

Chair(s)

TBA

A Parallel Algorithm Development Model for the GPU Architecture

J. Steven Kirtzic, **Ovidiu Daescu**
Department of Computer Science
University of Texas at Dallas
Richardson, TX USA
{jsk061000, daescu}@utdallas.edu

Abstract— *Parallel computing has been in use for decades, and throughout many researchers have sought to define a model for algorithm design for such a platform. Valiant developed a model for parallel computing, which was later extended to later include multi-core processors, but it still may not be best suited for the unique GPU architecture. With the current advances in high performance computing, it is easy to see the role that GPUs can play, and even easier to see the need for a model for GPU algorithm development. Here we propose a parallel GPU model which offers both a general design and a fine-grained approach, intended to accommodate nearly any GPU architecture. We show how our model can result in significant increases in performance when algorithms are designed based on its principles.*

Keywords: GPU, parallel processing, algorithm design

1. Introduction

The rapid advancement of the Graphics Processing Unit, or *GPU*, over the last few years has opened up a new world of possibilities for high-speed computation, ranging from biomedical to computer vision applications. Recent examples include [1], [2], and [3]. However, the GPU architecture is unlike that of any other, and designing algorithms to fully harness the capabilities of a GPU is not an easy task, especially when one considers the advantages and disadvantages of the various resources that a GPU has available to it. In this paper we introduce a parallel algorithm design model for the GPU architecture which addresses these issues. In Section 2 we discuss related work; in Section 3 we present a brief overview of the GPU architecture, focusing on NVIDIA's CUDA architecture; in Section 4 we present the model in its entirety; in Section 5 we illustrate the use of our model as we apply it to template and shape matching algorithms; in Section 6 we discuss the results of our model as applied to these template and shape matching algorithms; and finally in Section 7 we conclude and remark on future work.

1.1 Contribution

We believe that our main contribution with this work is to provide an easily accessible parallel algorithm design model for the GPU architecture. Our model addresses the limitations of other parallel models in that it accounts for the unique architecture of the GPU, in particular the various types of memory that the GPU possesses and their individual attributes. Our model is also designed to include single or multi-core CPUs as part of the system, if the designer chooses to do so. Finally, our model is intended to be easily accessible for a wide variety of reseachers from all scientific fields interested in GPU algorithm design, ranging from the novice to the experienced.

2. Related work

We present the following parallel model designs in succession to demonstrate the evolution of our Parallel GPU Model (PGM) and give it proper context.

2.1 The PRAM model

The PRAM model is generally regarded as one of the original parallel algorithm design models. The main shortcoming of the PRAM model lies in its unrealistic assumptions of zero communication overhead and instruction-level synchronization. Another drawback of with the PRAM model is that the time complexity of a PRAM algorithm is often expressed in big-O notation, which is often misleading because the machine size n is usually small in existing parallel computers. Consequently, the PRAM model is generally not used as a machine model for real-life parallel computers.

2.2 The BSP model

The BSP, or bulk-synchronous parallel model, was proposed by Leslie Valiant [4] to overcome the limitations of the PRAM model [5], while maintaining its simplicity. In the BSP model, a BSP computer consists of a set of n processor/memory pairs (nodes) that are interconnected by a communication network. The BPS model is *Multiple Instruction Multiple Data* (MIMD) in nature, and uses the concept of a *superstep*, which is comprised of a computation step, a communication step, and a synchronization step. The BSP model is also variable grained, loosely synchronous, has non-zero overhead, and uses message passing or shared variables for communication.

The program executes as a strict sequence of supersteps. In each superstep, a process executes the computation operations in at most w cycles, a communication operation that

takes gh cycles, and a barrier synchronization that takes l cycles. Note that in the communication overhead gh, g is the proportional coefficient for realizing a h relation. The value of g is platform-dependent, but independent of the communication pattern. In other words, gh is the time that it takes to execute the most time-consuming h relation.

Within a superstep, each computation operation uses only data in its local memory. This data is put into the local memory, either at the program start-up time or by the communication operations of previous supersteps. Therefore, the communication operations of a process are independent of other processes.

The BSP model is more realistic than the PRAM model because it accounts for all overheads except for the parallelism overhead for process management. The time for a superstep is estimated by the sum

$$w + gh + l \tag{1}$$

This model is highly regarded and has formed the basis for other parallel models, such as the parallel phase model [5], which we will briefly discuss next. However, its generality is its shortcoming when one attempts to apply it to more specific architectures, such as that of the GPU. Valiant recently extended his model to include multi-core CPUs [6]. While this model is much more akin to the architectural nature of the GPU, it still does not take into consideration the complexities of the typical GPU architecture, in particular the various types of memory, which as we will demonstrate in later sections have a tremendous impact on the performance of a given GPU algorithm.

2.3 The parallel phase model

Kai Hwang and Zhiwei Xu [7] proposed a phase parallel model for parallel computation that is further refined from the above two abstract models. This model is similar to the BSP model with the following distinctions: a parallel program is executed as a sequence of phases: the parallelism phase, the computation phase, and the interaction phase. The total execution time of the superstep on n processors is expressed by

$$\begin{aligned} T_n &= T_{comp} + T_{interact} + T_{par} \\ &= (w + \sigma\sqrt{2logn})t_f + t_0(n) + \alpha * w * t_c(n) + t_p(n) \end{aligned} \tag{2}$$

where w is the number of cycles, as with the BSP model, α is the *communication-to-communication ratio* (CCR) of each superstep, and t_f is the average time to execute a flop by a processor.

Improved from the PRAM and the BSP models, the phase parallel model is closer to covering real machine/program behavior. In this model, all types of overheads are accounted for, as shown in Eq. (2): the load imbalance overheads, the interaction overhead (t_0 and t_c terms), and the parallelism overhead (t_p term).

While these models represent the evolution of parallel algorithm design in general terms, they are limited in scope as they ultimately fall short when applied to the unique architecture of the modern GPU. The need for a model suited to this architecture was vocalized in a paper from MIT [8] in which the authors identify that official documentation for CUDA from NVIDIA was rather sparse, the forums required a lot of searching to find an answer to a particular problem, and the trade-offs between various programming options were difficult to discern. We attempt to address these issues by providing a model which was designed to not only include the more general models identified above, but to also take into consideration the unique nature of the GPU architecture, as it differs considerably from the CPU architecture.

3. GPU architecture

In this paper we will often refer to the machine containing the GPU as the "host" and the GPU itself as the "device". The NVIDIA GeForce 8800 series is an example of a typical GPGPU (General Purpose GPU) device, which utilizes NVIDIA's CUDA (*Compute Unified Device Architecture* GPU design. The GeForce 8800 contains 16 multiprocessors, each containing 8 semi-independent cores for a total of 128 processing units. Each of the 128 processors can run as many as 96 threads concurrently, for a maximum of 12,288 threads executing in parallel. The computing model is SIMD (*Single Instruction Multiple Data*), and the memory model is NUMA (*Non-Uniform Memory Access*) with a semi-shared address space. This stands in contrast to a modern CPU, which is typically either SISD (*Single Instruction Single Data*) or MIMD, in the case of a multi-processor or multi-core machine. Additionally, from the perspective of the programmer, all memory is explicitly shared (in multi-threading environments) or explicitly separate (in multi-processing environments) on a desktop machine.

3.1 GPU instruction throughput versus memory access

The GPU architecture is much more optimized for performing calculations than for memory accesses. Therefore, considering the multiple types of memory that the GPU architecture typically includes, it is important to keep this in mind when accessing these types of memory, particularly the slower, off-chip ones such as the GPU's global and the host's main memory. The most costly memory access is by far the host-to-device (CPU to GPU) data transfer, and reducing that transfer can have a tremendous impact on the overall performance of any algorithm that is implemented in part or fully on a GPU.

As an example of our research, we present the case of a typical Full Search Method of template matching, which is otherwise known as a "brute force" method. A naïve GPU

implementation of this algorithm is relatively easy, as the underlying architecture (such as CUDA) will handle most of the scheduling, thread allocation, memory management, etc. for you. In this case, a naïve, straightforward GPU implementation should run in *O(mn/p + log m)* time, where *p* is the number of processors, assuming that $1 \ll n$. We present the results of the implementation of this naïve GPU algorithm using several different types of GPU memory (discussed below) versus the serial implementation in Table 1.

Table 1: Run time for Full Search Method template matching for a 512x512 image and a 64x64 template. Times are in ms.

	Run Time	Copy Time
CPU	23290	N/A
GPU	3042	217.7
GPU Shared Memory	200.68	217.7
GPU Texture Memory	107.38	2.361

Table 2: Average results over 1000 trials of basic CUDA memory operations. The first column refers to the amount of data used for this experiment, in bytes. "malloc" and "malloc 2D" refer to allocating an array and a byte aligned 2 dimensional array on the GPU, respectively. "copy" and "copy 2D" refer to copying data from the CPU's global memory to the GPU's global memory. All times are in ms.

size	malloc	copy	malloc 2D	copy 2D
$4 * 10^3$	0.067567	0.005253	0.116700	0.014929
$4 * 10^5$	0.118616	0.291486	0.122187	0.296680
$4 * 10^6$	0.141160	2.576290	0.180513	2.713126
$4 * 10^7$	0.241793	23.344471	0.629537	24.801236

Given the considerable differences in architecture between the GPU and CPU, one can see that the ratio of overall run-times of the CPU to naïve GPU implementation (which we define as "speedup", *S*) is only 23290/3042 = 7.66. Given the number of processing cores *p* in our GPU is 128, this is clearly not an optimal solution, as it yields an efficiency of .060 (we define ''efficiency" as *E = S/p*). The majority of this is due to communication overhead (data transfer), as global memory on the GPU is uncached. Experimentation confirms that the instruction throughput is only .034, indicating that 96.6%≈97% of the total run time was due to host-device data transfer.

In addition to the host-device memory read/write, there are several other types of memory that a GPU may have and access, either on-chip or as part of the graphics card, including (in order of typical size) global memory, L2 cache memory, texture memory, shared (local) memory, and the processor registers. The type of memory that a programmer uses for a particular operation depends upon the size and nature of the data structures to be used, whether or not these data structures can be broken up (and if so how), and whether they are read/write data or simply read-only. In Table 2 we present the results of our experimentation with host-device data transfer times for different sizes of data to illustrate the importance of proper data partitioning.

4. Parallel Algorithm Design Procedure for GPUs

As discussed earlier, Valiant's multi-core parallel algorithm model falls short when one attempts to apply it to many-core GPUs. In designing our Parallel GPU Model, we opted to refer back to Valiant's original BSP model as a basis, and build out our model from there.

4.1 General model

GPU Superstep:

$$max_{i=1}^{p} w_i + max_{i=1}^{p}(h_i * g) + l \tag{3}$$

where p = number of processors (cores) on the GPU

w_i = cost of local computation in process *i*

h_i = number of messages sent and received and/or variables accessed by process *i*

l = cost of synchronization

g = message speed (bandwidth)

Algorithm total cost:

$$\begin{aligned} \mathrm{C}_{GPU} &= \sum_{i=1}^{S} superstep_i \\ &= W + H * g + S * l \\ &= \sum_{s=1}^{S} w * s + (\sum_{s=1}^{S} h_s)g + S * l \end{aligned} \tag{4}$$

where *S* = number of supersteps

W = total cost of local computations in all processes

H = total number of messages sent and received and/or variables accessed by all processes

CPU Superstep:

The CPU component of this general model is very similar to the GPU component above, with the exception that the variables apply to the CPU (i.e. *p* applies to the number of CPU cores, processes are executed on the CPU cores, etc.)

Total

$$C_{total} = \sum_{i=1}^{n} C_{GPU_i} + \sum_{i=1}^{n} C_{CPU_i} + (\sum_{i=1}^{m} T_i)b \tag{5}$$

where C = cost

n = number of algorithms or parts of algorithms executed

m = number of data transfers between CPU and GPU (usually an even number)

T = one-way data transfer

b = CPU to GPU data transfer bandwidth

In the simplest, single algorithm situation, this can be represented as:

$$C_{total} = C_{GPU} + C_{CPU} + 2 * T * b \quad (6)$$

Reducing the number of *T*s along with the size of *T*(the amount of data transferred) are two coarse-grained methods of reducing an algorithm's overall run time and thereby increasing performance.

4.2 Fine-grained model for parallel GPU algorithm design

If the algorithm designer is experienced in parallel algorithm design principles, they can typically skip to Step 3 below, otherwise they should continue with the following steps.

Step 1: Once a serial implementation of a given algorithm has been either acquired or originally designed, the first consideration is what types of instructions (operations) are to be performed on the given data set or sets of the algorithm. This will help in determining data dependency (as discussed below), as well as to help determine what operations must be performed on which processor.

Step 2: The goal is to reduce the total data transfer time as much as possible, meaning reducing the amount of data that is transferred back and forth between the host and device. Furthermore, this round trip may be performed multiple times for one algorithm depending on the structure of the algorithm and/or the size of the data sets. The various types of GPU memory are of varying sizes, but are all generally very small compared to modern host main memory. Therefore, in some parallel applications a data set may have to be broken up and sent to the GPU for computation through several round-trips. Consequently, as discussed briefly in Section 3.1, sending large amounts of data to and from the GPU unnecessarily may lead to a longer run time than the original serial algorithm (especially if it is optimized and/or implemented on a multi-core CPU).

With this type of trade-off of data and operations between the CPU and GPU, the most important aspect in determining what should be transferred to the GPU is *data dependency*. This essentially refers to identifying what operations require the data that is the result of previous operations. If an operation must wait for the resulting data from a previous operation, then these operations must be performed serially. They can still be performed serially on the GPU, but that would defeat the purpose of using the GPU and would require unnecessary data transfers. Furthermore, in most cases the CPU would be able to complete these serial operations faster than a GPU would, even without taking into consideration the costly host-device transfer time.

At this point, more experienced parallel GPU algorithm designers can proceed to Step 4 where we discuss optimizations specific to the GPU architecture. Otherwise it is recommended to take the following step of designing and implementing a naïve parallel algorithm before continuing to the optimization step.

Step 3: Implementing a naïve parallel GPU algorithm is a relatively straight-forward task. Most GPU architectures, such as CUDA, include a thread scheduler which will automatically distribute computations to threads and handle other high level functions of the GPU for you. What this results in is a simple port of a serial algorithm to a GPU with little or no consideration for the various aspects of a GPU's architecture that can be leveraged to create an optimal parallel algorithm. While a naïve parallel implementation can be accomplished rather quickly and easily resulting in a notable speedup of the algorithm's performance, this speedup will not be as great as it could be when the GPU architecture optimizations are performed, as discussed below.

Once the data dependencies within the algorithm have been identified, the designer is then able to break the up algorithm into the various parts that have to be done in serial and the parts that can be done in parallel. From that point, the designer can implement the serial parts on the CPU using typical CPU code (i.e. C/C++, Java, etc.) and implement the parallel parts using GPU code (i.e., C for CUDA, Brook+, etc.). With architectures such as CUDA, you can implement the CPU and GPU code in the same program, with the GPU code written simply as individual kernels that are called from within the CPU code, which simplifies the writing of the code a great deal. Also, recently many CPU and GPU manufacturers have adopted a language known as OpenCL, which allows algorithms to be implemented in a single language that can run on both the CPU and GPU, eliminating the need for separate languages for each architecture [9].

As far as GPU memory manipulation goes, with a naïve implementation it is usually easy to simply load the data set (or as much as possible at one time) into the GPU's global memory. The GPU will handle transferring the data from the host's RAM into the GPU's memory whenever a kernel is invoked. The designer just specifies, in the case of CUDA for example, which memory type is being used for which kernel when the kernel is defined in the code. The global memory, while being the largest type of memory and read-write capable, is also the slowest memory. Therefore it is one of the first areas to avoid, if possible, when performing algorithm optimizations, as is described in the next step.

To achieve optimal or near-optimal performance, we must take into consideration the unique architecture of the GPU and exploit this architecture to its fullest. It may take even the most experienced GPU algorithm designer several attempts to achieve optimal results, as often time optimality is best determined through experimentation. But careful analysis of the algorithm along with the GPU architecture can help to greatly reduce the need for experimentation to achieve optimality.

One basic yet effective step toward optimality is to eliminate unnecessary computations. As discussed above, naïve brute force parallel algorithms simply perform the same computation on an entire data set in one step (or several steps depending upon the size of the data set and the number of available threads). Our implementation of a naïve template matching algorithm searched the entire image for a match to the template, which required the transfer of the entire image data from the host's main memory to the GPU's global memory, a very costly operation. However, by redesigning our algorithm to first perform a "pruning step", we were able to reduce the overall runtime by as much as 99%, depending upon the amount of noise in the original image [12]. This is an example of placing "smart" bounds on the dataset to greatly reduce the amount of data that has to be transferred from the host to the device.

The next step in creating optimal parallel algorithms for the GPU is to determine what type and size of data structures your algorithm will use. These considerations are closely associated with what types of memory the algorithm will utilize, both within and outside of the GPU. As discussed in Section 3.1, the GPU architecture is unique in its design and varies considerably from that of a CPU. Indeed, it's fair to say that a GPU is analogous to being "a computer within a computer". Thus the various types of memory that a GPU contains and has access to certainly complicates considerations when designing parallel GPU algorithms. Following, we will discuss the various types of CPU/GPU accessible memory and their advantages/disadvantages.

4.2.1 GPU memory considerations

We denote a system's RAM as M, and note that it has the following attributes: it is read/write capable, is the largest sized memory overall (typically in the order of GB by current standards), its transfer speed (host to device) is the slowest by far, and it is not directly accessible by kernel threads. Global memory, which we denote with G has the following attributes: it is read/write capable, is the largest GPU (on card) memory, its transfer speed is the slowest for a given GPU and graphics card, and it is accessible by all threads. The L2 cache, denoted by L, is an example of a high capacity, high speed component that may or may not be available on a particular GPU, depending upon the model that one is using, such as in the case of the Fermi architecture (see [10]). The texture memory, which we denote as x, has the following attributes: it is read only, is smaller than global memory, but larger than shared memory, it is much faster than global memory but not as fast as shared memory, and is accessible by all kernel threads. The next type of memory is the shared local memory/L1 cache (in the cases where shared local memory also includes an L1 cache [10]). Shared local memory has the following attributes: it is read/write capable; is much smaller than texture memory but larger than the registers; and is somewhat faster than texture

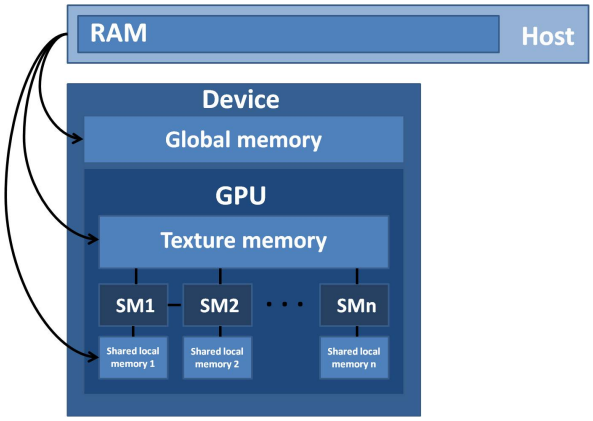


Fig. 1: The host(CPU)-device(GPU) memory hierarchy available to programmers (note that registers are excluded as they are not directly addressable by GPU programmers). We can see how the GPU must go through the host to load data from the RAM into the GPU's global, texture, or shared memories, the latter of which can communicate through SMs (Streaming Multiprocessors).

memory, but is accessible only by threads on a Streaming Multiprocessor (SM). Finally, we denote the register usage by$r$, where registers have the following attributes: they are read/write capable, they are the smallest and fastest of all memory types, and they are typically one register per core. We illustrate the various types of memory available to the GPU programmer with their relative sizes in Figure 1.

With this in mind, however, we will omit the following terms from the final PGM for the following reasons: the RAM transfer time (i.e. host-to-device) is accounted for by the term Tb in Equation (1). Further, we consider the fact that the registers are simply used by the cores as "scratch pads" to temporarily hold the data to be used by the processors and thus are not able to be directly manipulated by the programmer. Therefore, with the exception of bank conflict concerns, the register use should not be of consequence to the designer, and can then be eliminated from consideration.

Naturally, we wish to use the fastest type of memory available in all occasions. However, due to limitations imposed by the size and nature (i.e. read only or read/write) of the data structures that we use for a given algorithm, along with the sizes of the various types of memory as discussed above, a designer may choose different types of memory for various data structures. For example, with our template matching algorithm, we chose to use the texture memory to load our initial query image into, due to the fact that the texture memory is large, fast, and the image does not require write capabilities [12].

Obviously, with the goal being to take as much advantage of the fastest types of memory as possible, a designer's data structures may need to be altered to adhere to the size

limitations of certain forms of memory. As an example, if image data can be streamed into the shared memory blocks at a faster rate than transferring the entire image into global or even texture memory, (without disrupting the computation being performed in those blocks) then using the faster shared local memory is preferable.

Once the algorithm designer has an understanding of the various types of memory that the GPU provides, they can proceed to design the data structures that their parallel algorithm utilizes accordingly. This includes deciding how to break up the data itself, so that the size and the nature of the data structures allow them to be grouped into logical blocks that can be mapped into thread blocks of one, two, or three dimensions, where the threads can maximize their intercommunication through variable sharing within the SM's shared local memory. With our GPU template matching algorithm (discussed in Section 5), we demonstrate the advantages of not only breaking up of the data into logical blocks, (or "strips" in this case) but also the smart use of the GPU's texture memory instead of the GPU global memory, with the former being much faster.

It is important at this point to reinforce that the naïve parallel implementation will typically be subject to the underlying system's automatic (typically non-optimal) scheduling system, which will typically distribute threads on a first-come-first-serve basis, which is generally not optimal.

Another important technique toward achieving optimality with a given parallel GPU algorithm is identifying and reducing (or even eliminating) algorithm execution bottlenecks. As we have discussed above, the execution bottlenecks primarily are memory transfers, in particular the host to device transfer. This issue is being addressed with the new generation of CUDA architecture (Fermi [10]), but for the present the average algorithm designer and implementer dealing with commodity GPUs needs to consider the massive host-device bottleneck.

Above we discussed several ways to reduce the amount of data transfer between the host and device, as well as ways to design your data structures to fully take advantage of common GPU architectures. Following we shall formally define our Parallel GPU Model based upon the above considerations.

Previously, we defined a GPU operation in the following manner:

$$\begin{aligned} C_{GPU} &= \sum_{i=1}^{S} \text{superstep}_i \\ &= W + H * g + S * l \\ &= \sum_{s=1}^{S} w_s + (\sum_{s=1}^{S} h_s)g + S * l \end{aligned} \tag{7}$$

where S = number of supersteps, W = total cost of local computations in all processes, H = total number of messages sent and received and/or variables accessed by all processes, w_s = cost of local computation in process s, h_s = number of messages sent and received and/or variables accessed by process s, l = cost of synchronization, and g = message speed (bandwidth)

Generally parallel algorithms work toward computing a result from a large number of simpler calculations performed in parallel on the various processors/cores in the system. This requires a reduction step, which as discussed above typically runs in O(log *m)* time. We represent this step as R and add it to Equation (7), which gives us:

$$\sum_{s=1}^{S} w_s + (\sum_{s=1}^{S} h_s)g + S * l + R \tag{8}$$

The PGM is essentially an extension/adaptation of existing parallel algorithm models, including the PRAM, BSP, and Parallel Phase Model. However, our model is focused on the GPU architecture, which requires the redefinition of certain terms from the original BSP/Parallel Phase Model. In our model, we equate a BSP/Parallel Phase Model superstep with the execution of a GPU "kernel", which is essentially a GPU function or method which handles the importing of the data set, the computations to be performed on said data set, and the exporting of the resulting data to the CPU (i. e. RAM). Therefore we shall now denote a kernel/superstep as k, with the total number of kernels executed as K. This gives us an updated version of Equation (8) as:

$$\sum_{k=1}^{K} w_k + (\sum_{k=1}^{K} h_k)g + S * l + R \tag{9}$$

4.2.2 Message/variable passing

The term $(\sum_{k=1}^{K} h_k)g$, which describes the total number of messages and/or variables transmitted multiplied by the bandwidth of the transfer medium, is perhaps the most important of the terms and deserves more consideration. As we expand this term to include the various types of memory that a GPU can read from and write to, we get

$$\begin{aligned} (\sum_{k=1}^{K} h_k)g =& (\sum_{a=1}^{A} h_a)G + (\sum_{b=1}^{B} h_b)L \\ &+ (\sum_{c=1}^{C} h_c)x + (\sum_{d=1}^{D} h_d)y \end{aligned} \tag{10}$$

where a = an individual global memory read/write
A = the total global memory reads/writes
b = an individual L2 cache memory read/write
B = the total L2 cache reads/writes
c = an individual texture memory read/write
C = the total texture memory reads/writes
d = an individual shared memory read/write
D = the total shared memory reads/writes

When we substitute this more accurate representation of memory reads/writes into Equation (2), we get:

$$\sum_{k=1}^{K} w_k + (\sum_{a=1}^{A} h_a)G + (\sum_{b=1}^{B} h_b)L + (\sum_{c=1}^{C} h_c)x + (\sum_{d=1}^{D} h_d)y) + S * l + R \quad (11)$$

It should be noted that with memory transfers we do not take into consideration the size of a word for the particular system, so we are utilizing the *uniform cost criterion*, as is commonly done with parallel algorithm design and analysis [11]. Also, based upon the ratios of run times between the global, texture, and shared local memories, we can calculate coefficients to represent the approximate cost of using each type of memory and add these coefficients to Equation (5). Normalized with respect to global memory we get ratios of 0.035 and 0.066 for the texture and shared memories, respectively. Note that these two values are roughly in a ratio of 1 to 2 in relation to each other. Applying this to Equation (5) then gives us the following:

$$\sum_{k=1}^{K} w_k + (\sum_{a=1}^{A} h_a)G + (\sum_{b=1}^{B} h_b)L + 0.035(\sum_{c=1}^{C} h_c)x + 0.066(\sum_{d=1}^{D} h_d)y + S * l + R \quad (12)$$

Which can be generalized to:

$$\sum_{k=1}^{K} w_k + (\sum_{a=1}^{A} h_a)G + (\sum_{b=1}^{B} h_b)L + \frac{1}{32}(\sum_{c=1}^{C} h_c)x + \frac{1}{16}(\sum_{d=1}^{D} h_d)y + S * l + R \quad (13)$$

Thus, with the above equation we can see the relative costs of using various types of memory addressable by the GPU programmer. This model is intended to be applicable to most, if not all GPU architectures: indeed, if a particular architectural feature is not available with the model of GPU being employed (i.e. L2 cache memory), then that term is simply zeroed out or removed from the above equation.

Step 5:The last step of this Parallel GPU Model design procedure is more advanced and involves a more intimate knowledge of the specifics of the architecture of the particular GPU the algorithm is designed for. It should be noted that this step is not necessary to achieve near-optimality, as that can typically be achieved by adhering to the above steps. However, for the designer desiring as much optimality as possible, they should identify and understand several physical aspects of the particular device. These physical aspects include the following: the number of processors/cores that the GPU has; the number of SMs the GPU has; the number of processors/cores per SM; the amount of global, texture, and shared local memory the GPU has; the type and speed of the connection between the host and the device (i.e. PCI, PCIe, etc.); the availability and size of the L1 and the L2 caches (as with Fermi GPUs); whether or not the particular GPU architecture supports IEEE 754-2008 (which includes full double precision support); the nature of the GPU's warp scheduler (i. e. whether or not it is a single or dual warp scheduler); whether or not the device has Error Correcting Code (ECC) memory support; whether the device is 32 or 64 bit-based; and what programming languages the device supports, such as C/C++ for CUDA, OpenCL, etc. (and to what degree).

By knowing hardware-specific details, such as the number of processors/cores in the particular GPU, we can augment our abbreviated PGM with the above substitution, which yields the following version of the PGM:

$$\sum_{k=1}^{K} \frac{mn}{p} + (\sum_{k=1}^{K} h_k)g + S * l + R \quad (14)$$

where *mn/p* = w_k
m = number of computations to perform
n = number of data elements
p = number of processors/cores

5. Applications of the PGM

We first applied our GPM to the field of template matching in [12]. Here we developed a GPU-accelerated template matching algorithm from the ground up based upon the PGM. Template matching essentially involves searching an image *I* attempting to find the match for a template *x* among all possible candidates y_i in a sort of "sliding window" fashion. The searches are independent of each other and therefore are highly subject to parallel processing on the GPU. Furthermore, we employed a pruning step to eliminate unnecessary data transfer to the GPU, as discussed in Step 3 of Section 4 above.

In our second application of the PGM, we chose to apply it to a shape matching algorithm that we had developed [13] previously. In this case, a 3D shape is given a unique "signature" which is calculated by computing the distances between each vertex of the shape and every other vertex, as long as that vertex is "visible" to the original vertex. The result is a large number of calculations which are then formulated into a histogram, which then forms the shape's unique signature. Obviously, these calculations are completely independent of each other and therefore this algorithm is also a very good candidate for parallel processing on the GPU.

6. Results and discussion

The experimental design for our template matching algorithm consisted of averaging the results of running our algorithm over a number of trials with a variety of images of different sizes and resolutions, which yielded the following performance results: when comparing the performance of our template matching algorithm to the Full Search Method discussed earlier on small images (512x512) at zero to low noise levels, our algorithm has better performance than the Full Search Method. When comparing our algorithm's performance to that of a standard brute-force Full Search Method implemented serially on the CPU on medium to large images one can see the tremendous performance increase of our algorithm. With an image size of 1024x1024 and a template size of 256x256, our algorithm experiences a 8700x performance increase over the Full Search Method. Further, when we implemented the Full Search Method in a naïve parallel manner on the GPU, our optimized algorithm performed 39x faster.

Similarly we observed a considerable speedup in run-time in our shape matching algorithm when applying the PGM to it. Serially, this algorithm has a run-time of:

$$O(n^2\ \lambda(n)log(n/\epsilon)/\epsilon^4 + n^2\ log(np)\ log(n\ log\ p)$$

However, with the application of the PGM, we observe the following: if the number of data items equals n and the number of processors equals p, then the total computation time for the above shape matching algorithm is:

$$n/p + logn \tag{15}$$

where *log n* is the reduction step. This is assuming that at each timestep a processor p is calculating the distance from a query point to another point. The reduction step results in the shortest distance from the query point to the signature point. Therefore, this algorithm could perform in near-linear time, depending upon the number processors in the parallel system.

In comparison with other parallel design models, we observe the following: the PRAM model, while being a fundamental and an "all-encompassing" parallel design model, is rather inadequate when applied to modern GPGPU design, due to its generality. Valiant's BSP model is a MIMD model consisting of node/memory pairs interconnected through a network. This is not very analogous to modern GPU architectures. The parallel phase model attempts to make up for this shortcoming by accounting for all overhead costs, but is still based on a model which does not account for the various types of memory that the GPU possesses, nor their individual advantages and disadvantages. Valiant's more recent multi-core model is much more akin to the many-core GPU architecture, however it still falls short when one considers the various types of GPU memory, including their attributes and design concerns. We believe that our model is well-suited to the GPU architecture by accounting for all possible types of memory and their associated costs that a GPU can access, both with current commodity GPUs and new, more advanced architectures. Furthermore, we provide a thorough analysis of the various considerations that one must keep in mind when designing algorithms for any GPU architecture, and we believe that our model provides an opportunity to do so that other models don't.

7. Conclusion and future work

Overall, we believe that our parallel GPU method is very effective in allowing parallel GPU algorithm designers, ranging from the novice to the expert, to design and implement optimal (or nearly optimal) algorithms that take advantage of the GPU architecture. We noted that while previous models have been adequate for general parallel architectures (which can vary considerably), they fall short when addressing the unique architecture of GPUs. Indeed, the degree of performance that can be achieved with the application of other parallel models, such as the BSP and phase parallel model, is not optimal for the GPU architecture, and we showed how the PGM can achieve a greater degree of optimality. Therefore, depending upon the degree of optimality desired, our model seems superior to other existing parallel models when applied to the GPU architecture.

Future work with our model will include applying it to simulations involving radiation therapy, such as Volume Modulated Arc Therapy (VMAT) and Intensity Modulated Radiation Therapy (IMRT). In such simulations, various computations need to be made in real-time or near real-time and the use of GPUs would certainly be a great advantage. Also, more practical experience with a Fermi GPU would allow for the implementation of various algorithms to quantitatively measure the speedup that a particular architecture allows over other parallel and serial architectures. Finally, the development of software which employs an "automated" version of this model would make it accessible to even more researchers by aiding them in making more informed with their algorithm designs, thereby producing even more optimal results.

References

[1] D. Qiu, S. May, and A. Nüchter, "GPU-accelerated nearest neighbor search for 3d registration," *Computer Vision Systems*, pp. 194–203, 2009.

[2] P. Noël, A. Walczak, K. Hoffmann, J. Xu, J. Corso, and S. Schafer, "Clinical evaluation of GPU-based cone beam computed tomography," *Proc. of High-Performance Medical Image Computing and Computer-Aided Intervention (HP-MICCAI)*, 2008.

[3] J. Huang, S. Ponce, S. Park, Y. Cao, and F. Quek, "GPU-accelerated computation for robust motion tracking using the CUDA framework," in *Visual Information Engineering, 2008. VIE 2008. 5th International Conference on*. IET, 2008, pp. 437–442.

[4] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990. [Online]. Available: http://portal.acm.org/citation.cfm?id=79181

[5] K. Hwang and Z. Xu, *Scalable parallel computing: technology, architecture, programming*. WCB/McGraw-Hill, 1998.

[6] L. Valiant, "A bridging model for multi-core computing," *Journal of Computer and System Sciences*, vol. 77, no. 1, pp. 154–166, 2011.

[7] Z. Xu and K. Hwang, "Early prediction of mpp performance: the sp2, t3d, and paragon experiences," *Parallel Computing*, vol. 22, no. 7, pp. 917–942, 1996.

[8] Massachusetts Institute of Technology, "IAP09 CUDA@MIT 6.963," MIT, 2009. [Online]. Available: http://sites.google.com/site/cudaiap2009/home

[9] "Opencl programming guide for the cuda architecture v2.3."

[10] NVIDIA Corp., "NVIDIA's next generation CUDA compute architecture: Fermi," Sept. 2009. [Online]. Available: http://www.nvidia.com/object/fermi_architecture.html

[11] J. Jaja, *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., 1992.

[12] R. Anderson, J. Kirtzic, and O. Daescu, "Applying parallel design techniques to template matching with gpus," in *High Performance Computing for Computational Science–VECPAR 2010: 9th International Conference, Berkeley, CA, USA, June 22-25, 2010, Revised, Selected Papers*, vol. 6449. Springer-Verlag New York Inc, 2011, p. 456.

[13] Y. K. Cheung and O. Daescu, "Approximate point-to-face shortest paths in ^{3}," *CoRR*, vol. abs/1004.1588, 2010. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1004.htmlabs-1004-1588

A GPU Support for Large Scale Quantum Chemistry Applications

Selva Kumar Sengottaiyan[1], **Fang Liu**[1], **Masha Sosonkina**[1]
[1] Ames Laboratory, Iowa State University

Abstract— *GPU/GPGPU computing has been used widely in scientific simulation to improve the performance on hybrid architectures. The quantum chemistry field has benefited greatly from using GPUs, including tasks such as visualization of molecular orbitals and computation of electronic structures. To gain significant success in using GPUs, a large amount of code rewriting and restructuring is required, which is done primarily by those who understand the algorithm in great detail. In this paper, two widely used quantum chemistry packages are investigated to identify the hot spots that can benefit most from GPUs, as well as be the least intrusive to the existing code base. The paper uses an experimental approach to integrate GPU capability without restructuring the application. Experimental results show that the bottleneck is in CPU–GPU data transmission. Additionally, a GPU-based DFTFOCK method is implemented in GAMESS/NWChem and a GPU-based eigensolver is integrated with NWChem successfully. Further performance tuning is ongoing.*

Keywords: GPU Computing, Eigensolver, Quantum Chemistry, GAMESS, NWChem

1. Introduction

Graphics processing unit (GPU) computing or general-purpose computing on graphics processing units (GPGPU) is the use of a GPU to do general purpose scientific and engineering computing. The model for GPU computing is to use a central processing unit (CPU) and GPU together in a heterogeneous co-processing computing model. While one part of the application runs on the CPU, another computationally intensive part can be accelerated by the GPU. GPUs are an excellent accelerator for data-parallel applications. From the user's perspective, the application just runs faster because it is using the high performance of the GPU to boost performance. Large gains in performance have been achieved through GPUs in recent years and GPUs have become ubiquitous in handhelds, laptops, desktops, and supercomputer clusters. The power of GPUs has been incorporated into simulations or experiments and shows a large benefit. However, a large amount of code rewriting and restructuring is often required. GPU computing has recently begun to be adopted in the quantum chemistry domain [7] from visualization of molecular orbitals to computation of electronic structures.

The General Atomic and Molecular Electronic Structure System (GAMESS) is a widely used computational chemistry package [3], [8] for *ab initio* molecular quantum chemistry. Using GAMESS, a variety of molecular properties, ranging from simple dipole moments to frequency dependent hyperpolarizabilities may be computed. GAMESS is capable of a very broad range of electronic structure theory calculations and is therefore very widely used, with an estimated user base of $150,000$ in more than 100 countries.

NWChem [12] is a high-performance computational chemistry software package that focuses on providing new and essential scientific capabilities to its users in the areas of the kinetics and dynamics of chemical transformations. Initially, the problems of interest focused on environmental issues, but recently NWChem has been applied to the examination of metal clusters, biological systems, nanostructures, and materials. Both GAMESS and NWChem offer a multitude of highly correlated methods, density functional theory (DFT) with many exchange correlation functionals. Additionally, NWChem provides plane-wave DFT with exact exchange and Car-Parrinello simulations, molecular dynamics with the AMBER and CHARMM force fields, and combinations of these methods.

GPU and CPU hybrid platform have become widely used during the past few years. Well designed algorithms can fully take advantage of the performance of GPUs. Since quantum chemistry applications such as GAMESS and NWChem have always been at the forefront of improving time to solution for platforms from desktops to supercomputers, it is natural for these codes to use the computing power of GPUs.

In this paper, after briefly introducing the GPU computing model, two GPU eigensolver packages are compared. Next, details of the integration work are given, followed by the experimental results, related work and conclusions.

2. GPU computing model

The success of GPGPUs in the past few years has been due in part to the ease of programming of the associated CUDA parallel programming model. In this programming model, the application developers modify their application to take the compute-intensive kernels and map them to the GPU. The rest of the application remains on the CPU. Mapping a function to the GPU involves extensively rewriting the function, usually in C or C++, to expose the parallelism in the function and adding keywords to move data to and from the GPU. The developer is tasked with launching $10s$ to $1000s$ of threads simultaneously. The GPU hardware manages the threads and does thread scheduling.

Figure 1 shows the GPU computing model in which the GPU is a compute device which serves as a co-processor for the host CPU. The GPU architecture consists of a scalable number of streaming multiprocessor (SM)s, a multi threaded instruction fetch and issue unit, and a read-only constant

cache. A SM consists of Scalar Processor (SP) cores, special function units for transcendentals, a multi threaded instruction unit, and on-chip shared memory. The SM creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. To manage hundreds of threads running several different programs, it employs a new architecture single-instruction, multiple-thread (SIMT). The SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.

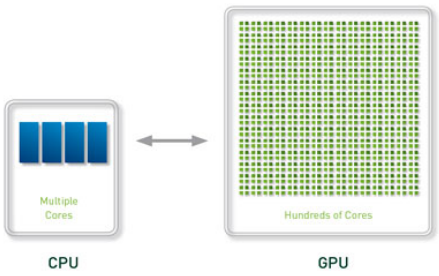


Fig. 1: GPU computing model

a) Threads: For GPU acceleration, a program must be decomposed into a large number of concurrently schedulable units so that groups of threads can execute the same code parallel to each other or suffer poor performance. In GPUs these groups, known as warps, consist of 32 threads. If threads within a warp diverge on a branch, the full warp is serially executed on each branch path, with threads converging into a single execution path only after the divergent branch is finished. CUDA also requires the organization of warps into larger units called blocks. Threads are assigned in units of blocks and can only communicate directly with other threads in the same block. Communication across blocks requires termination of the GPU kernel and data transfer into CPU memory where the required data manipulation can be performed. These issues limit the applicability of GPUs primarily to data parallel applications.

b) Memory hierarchy: Each thread has a private local memory. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global GPU memory. The memory model gives the developer the freedom to choose global or local memory, which may affect the performance.

3. GPU eigenvalue computing packages

There are some eigensolver packages with GPU support which are publicly available. The GPU-accelerated linear algebra libraries (CULA) and Matrix Algebra on GPU and Multicore Architectures (MAGMA) packages utilize the NVIDIA CUDA parallel computing architecture to dramatically improve the computational speed of sophisticated mathematics.

CULA[1] is a high-performance linear algebra library that executes in a unified CPU/GPU hybrid environment. CULA provides easy interfaces through which an application can be integrated without extensive GPU programming experience. CULA can provide significant speedups over existing packages and supports both dense and sparse linear algebra. CULA features a wide variety of linear algebra functions, including but not limited to, least squares solvers (constrained and unconstrained), system solvers (general and symmetric positive definite), eigensolvers (general and symmetric), singular value decompositions, and many useful factorizations (QR, Hessenberg, etc.). All such routines are presented in four standard data types in the Linear Algebra PACKage (LAPACK) computations: single precision real (S), double precision real (D), single precision complex (C), and double precision complex (Z).

The **MAGMA** project[2] aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current "Multicore+GPU" systems. MAGMA provides linear algebra algorithms, designs and frameworks for hybrid many core and GPU systems that can enable applications to fully exploit the power that each of the hybrid components offer. This package also aims to solve a nonsymmetric linear system of equations by increasing speed for the price of relaxed accuracy.

a) Comparison between the two packages: CULA supports both dense and sparse linear algebra while MAGMA mainly supports dense linear algebra. CULA supports both single and double precision mathematics, and MAGMA supports only single precision. The eigensolver routine in CULA is a commercial code, while MAGMA is free. Both packages have multi-GPU support.

Based on requirement in NWChem,where single precision is sufficient. We have chosen MAGMA for our first stage of integration.

4. Integration of GPU support

The DFT algorithms in GAMESS and NWChem are integrated with GPU support, while the GPU-based eigensolver, MAGMA, is linked to NWChem. The experimental work examined these two well-known packages and identified the hot spots for GPU integration.

4.1 DFT integration

a) GAMESS DFT with GPU: In GAMESS, the typical DFT approach solves the Kohn-Sham equation [6] in which the total energy of the molecular system is a function of the positions of the atoms and one-particle densities. The approach in DFT is to assume an initial charge density and obtain successively better approximations of the density and energy. When the total energy is minimized with respect to the variational parameters, the resulting one-particle

[1]CULA tools `http://www.culatools.com`
[2]MAGMA `http://icl.cs.utk.edu/magma/index.html`

equations are exactly the same as the Hartree-Fock method except for the handling of the exchange terms and the way the electron exchange correlation is incorporated. DFT methods can yield results similar to those obtained with *ab initio* methods such as MP2, but at a substantially reduced computational effort.

The major hot spot in a GAMESS DFT energy and gradient calculation is in a routine called 'GRDDFT' which calculates the correlation correction to Self-Consistent Field (SCF) with an arbitrary set of density functionals. The calculation of 'GRDDFT' takes 94% of the total DFT calculation time. The routine consists of four parts:

1) Memory allocation
2) Geometry and symmetry setting (DFTSET).
3) Calculating the exchange correlation energy (DMATD).
4) Calculating the exchange correlation energy gradient (DFTGRAD).

The calculation of the energy exchange correlation matrix takes almost 99% of the total GRDDFT execution time. The function DMATD calculates the exchange correlation energy by looping over radial grids which in turn loop over the angular grids surrounding atoms. The looping over radial grids and angular grids takes almost 99% of the total DMATD execution time. Inside the loop over grid points, DFTFOCK (which adds the DFT exchange/correlation contribution to the Fock matrix) takes the largest amount (72%)of the execution time. Thus DFTFOCK is chosen as the routine to be executed on the GPU. Other reasons are that the data dependency of DFTFOCK as compared to other subroutines is minimal and the amount of parallelism inside the function is high.

Figure 2 shows the integration of the CPU Fortran code with the GPU Fortran code in GAMESS. Inside the GAMESS Fortran code, the application code that has a high degree of parallelism and that takes most of the CPU time is identified. Then the identified code is transfered into CUDA Fortran. In the above case, the DFTFOCK subroutine is identified as a hot spot and converted into CUDA Fortran. Then the changed code is compiled using the CUDA Fortran compiler (PGFortran). The compiled code is linked with GAMESS and the CUDA libraries.

b) NWChem DFT with GPU: NWChem contains a parallel implementation of the Hohenberg-Kohn-Sham formalism [4] of DFT which differs significantly from other *ab initio* methods in the treatment of the exchange-correlation term used in building the Fock matrix. The computationally intensive components of a DFT calculation include the fitting of the charge density, construction of the Coulomb potential, construction of the exchange correlation potential, and the subsequent diagonalization of the resulting equations. The GPU accleration of DFT in NWChem concentrates getting exchange-correlation contribution to the gradient and adding the Bonacic-Fantucci repulsive term [6].

Figure 2 shows the integration of the CPU Fortran code with GPU Fortran code in NWChem. As with the GAMESS integration, the application code that has a high degree of parallelism and that takes most of the CPU time is identified. Then the identified code is changed into CUDA Fortran. The NWChem GPU kernel is similar to GAMESS GPU kernel (Figure 2) as in both cases the GPU kernel uses the CUDA runtime API and CUDA libraries. During integration of the GPU kernel in DFT for both NWChem and GAMESS, the GPU kernel implementation requires only minimum code rewriting as using the existing code is one of objectives of this GPU acceleration implementation.

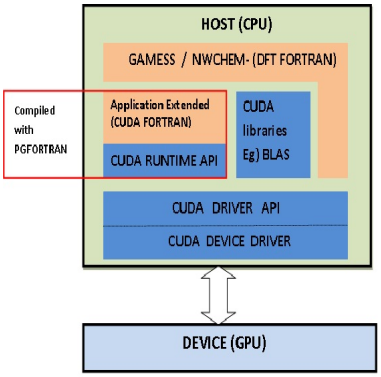


Fig. 2: GAMESS/NWChem DFT-GPU Programming

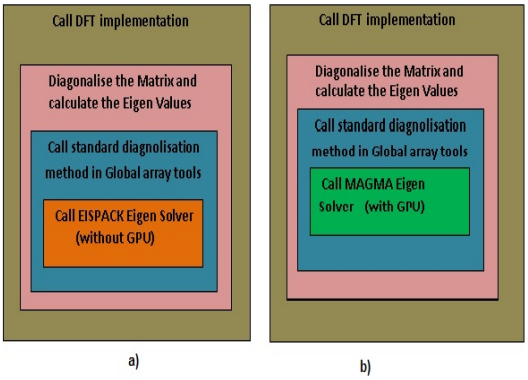


Fig. 3: Function calls inside the NWChem eigensolver implementation without and with GPU implementation

4.2 Integration of NWChem DFT eigensolver with MAGMA

Figure 3a shows the order of function calls inside the NWChem the DFT eigensolver DFT implementation without GPU implementation. Figure 3b shows the changed order of function calls inside the NWChem DFT eigensolver implementation with GPU implementation. Figure 4 shows the integration of the CPU Fortran code with GPU Fortran code (MAGMA eigenvalue package) in NWChem. The calculation of eigenvalues and eigenvectors is a time consuming part in the Fock matrix updating procedure. The MAGMA library of C functions offers two LAPACK-style interfaces, referred to as the GPU interface and the

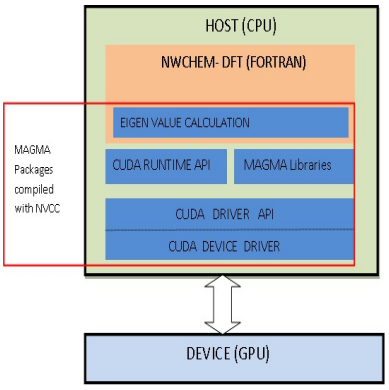


Fig. 4: NWChem-GPU MAGMA Interface

CPU interface. The GPU interface takes input and produces results in the GPU's memory, whereas the CPU interface produces results in the CPU's memory. The GPU and CPU interfaces, although similar, are not derivatives of each other, but instead have different communication patterns. The GPU function returns or sets the logical flag indicating whether the GPU interface is to be used for calculations involving the specified MAGMA matrix. NWChem normally calculates the eigenvalue by using the EISPACK available with the GA tool.

Global Array eigenvalue calculation

The Global Array diagnolization subroutine calls a sequence of subroutines from the eigensystem subroutine package (EISPACK) [10] to find the eigenvalues and eigenvectors (optional) of a real symmetric matrix. In NWChem for serial execution, the eigensolver 'rs' from EISPACK is used. (The eigensolver in ScaLAPACK [1] is called only for parallel execution.)

In order to demonstrate the GPU version eigensolver can benefit NWChem, the EISPACK eigenvector calculation in NWChem will be replaced by the MAGMA GPU solvers. Since NWChem invokes the 'rs' solver through the Global Array interface, the Global Array diagonalization subroutine EISPACK calls are replaced by MAGMA calls. The MAGMA eigensolver routine is Magma_dsyev which is very similar to the analogous LAPACK routine. The major difference between the 'rs' subroutine and Magma_dsyevd is that 'rs' uses the whole real symmetric matrix whereas Magma_dsyev uses the lower or upper triangular matrix. By switching from the 'rs' routine to the Magma_dsyevd, the matrix needs to be converted to a new format.

5. Experiments

All the tests are conducted at the US Department of Energy Ames Laboratory GPU computing cluster - Exalted, which consists of 26 nodes, each having six 2.66 GHz Intel Xeon processors, 8 nodes have 4 NVIDIA Tesla C2070 GPUs and 18 nodes have 2 NVIDIA Tesla C2070 GPUs. The nodes are connected via Mellanox QDR InfiniBand and each node has 24GB of RAM. Despite having two GPUs per node available, we focused on using only a single GPU per node. The speedup here is defined as the ratio of the existing sequential algorithm execution to the parallel execution time.

5.1 GPU implementation of DFT in GAMESS

This experiment on the DFT method in GAMESS focuses on performance improvement of the algorithm in terms of speed up. As explained in Section 4.1, the DFTFOCK routine is separated from the existing application and is tested with various problem sizes (Fock matrix sizes) to find the actual performance gain in terms of speed up. Various scenarios with different optimization techniques are tested and results are explained below.

Scenario 1: The GPU CUDA memory model comprises various memory spaces, which differ enormously in latency times, availability of caches, etc. Particularly important features are global and shared memories, the former being an off-chip memory and thus featuring high-access latency; the latter being on-chip memory and thus having reduced latency. This scenario tests the global and shared memory usage inside the GPU. The experiment results with problems of various matrix size are shown in Table 3. Results are after optimization for the number of threads and blocks based on the problem size. In order to find the break point where DFTFOCK can start to take advantage of the GPU, the matrix size is continually increased. The GPU implementation shows better timings for problems with matrix size of 10,000 or more (Table3). For the problem size of 20,000, there is not much difference in performance between CPU implementation and CPU-GPU implementation using global memory, but the CPU-GPU implementation using shared memory increases the speed up from the CPU implementation (Table 3, columns 1&3) 2 times and the CPU-GPU shared memory implementation also increases the speed up from the global memory implementation (columns 2&3) by about 2 times. The experimental results prove that the GPU implementation should involve less kernel calls and memory allocations for better performance. Shared memory is exploiting the availability of on-chip shared memory by enabling kernels to load all needed field components, including the components corresponding to adjacent blocks and provides huge performance gains in terms of GPU and CPU execution time. Such gains are due both to the high efficiency of shared-memory access, and due to the limited number of separate memory transaction issued for each thread i.e., reducing the effective number of memory read operations for each thread. GPU speed up can be increased more if memory usage is optimized. Hence it is necessary to carefully design and implement kernel codes in order to minimize the number of global-memory reads, by making use of the other available kinds of memory.

Scenario 2: In this scenario, we used various Fock matrix sizes to explain the factors affecting CPU-GPU implementation. An experiment (CASE A) is built with 50 outer loops and 12525 inner loop iterations. This computation takes about 20 microseconds in the CPU implementation. The total execution time for the CPU-GPU implementation is 70

Table 1: **Comparison of CPU, global memory GPU and shared memory GPU times in seconds (s)**

Size of Matrix	Total CPU execution time(s)	CPU+GPU execution time(Global)(s)	CPU+GPU execution time (Shared (s)
50	$20*10^{-6}$	$70 *10^{-3}$	$70*10^{-3}$
500	$100*10^{-6}$	$74 *10^{-3}$	$74*10^{-3}$
1000	$2.145237*10^{-3}$	4.213462	2.561576
5000	1.80	27.8	17.4
10000	211.2	277.0	173.6
20000	1732.1	1687.1	973.2

milliseconds. As expected, the GPU time is much greater than the CPU execution time. The breakdown of execution time for the CPU-GPU implementation (Table 2) shows that memory allocation time in the CPU-GPU code far exceeds the total CPU execution time. So experiments (B,C,D,E) are built with a problem size of 500 outer loops and 125250 inner loop iterations and execute with various GPU-CPU implementation/optimization techniques. The results are shown in Table 2. For all cases (B,C,D,E), each GPU block has 512 threads and the warp size is 32.

CASE B : Implementation of inner loop (DFTFOCK) in the GPU kernel, allocating the memory outside the kernel call. In this case, a small kernel size and large number of threads and memory allocation calls are needed in the GPU.
CASE C : Implementation of inner loop (DFTFOCK) in the GPU and allocating the memory inside the CUDA kernel.
CASE D : Implementation of outer loop (DFTFOCK) in the GPU kernel, allocation of memory outside the outer loop with a single kernel. In this case, the kernel size is increased but the number of threads inside the problem is decreased.
CASE E : Implementation of outer loop (DFTFOCK) in the GPU kernel, allocation of memory outside the outer loop and splitting the GPU kernel into two parts. In this case, the kernel size needed for implementation is increased and the number of threads inside the problem is decreased.

From Table 2, CASE A shows that the kernel call in a loop is the bottleneck as each kernel initialization takes a lot of time. Cases B, C, D and E also demonstrate that allocation takes more time than kernel execution, and therefore GPU implementation should involve less kernel calls and memory allocations.Kernel execution time is in micro seconds range where as total execution time is in milliseconds range. The best execution time is from the results of CASE D (i.e., the implementation involves less GPU memory allocations and less number of kernel calls irrespective of kernel size). Test case D has increased kernel size (i.e., more functions) compared to the kernels in other test cases, irrespective of which it almost takes same time as the other test cases. Thus, for GPUs, global inter-thread synchronization from kernel calls is very costly, because it involves a kernel termination and a new kernel call overhead from the host. The application specific software optimization is critical to fully utilize compute/bandwidth resources for both CPUs and GPU.

5.2 DFT acceleration using GPU in NWChem

The experiment on the DFT-FOCK method in GAMESS focuses on performance improvement of the algorithm in terms of speed. As explained in Section 4.1, the DFT GPU implemention is also done in the NWChem package. The problem size is determined by the number of atoms used in the input to the DFT NWChem package. The number of GPU threads inside each block is 512 and the warp size is 32. Table 3 shows the speedup in DFT using the GPU for various problem sizes. With the increase in problem size, the speedup increases. Due to data transfer latency, the benefit of using the GPU appears only if the problem size is increased to more than 2000 atoms. The size of the kernel is also very important in determing the speedup in the GPU.

Figure5, shows the running time for different numbers of threads for a problem size of 10000 atoms. Figure5 explains performance of GPU in terms of number of threads per block. The decrease in time with an increase in GPU occupancy (threads) shows that more performance gain is possible. A more occupancy indicates that the application fully exploits available processing units. Unfortunately, the amount of shared memory and registers used by each thread block limits the occupancy value. The size of thread blocks and/or shared-memory and registers usage must be designed with care in order to maximize the occupancy. The speedup increases with the increase in problem size but is greatly limited by data transfer between the CPU and GPU. The limitation in speedup highlights the importance of software optimizations (memory and GPU occupancy), and an application driven design methodology. s

5.3 MAGMA with eigensolver

The MAGMA package is integrated with NWChem package and the eigensolver of EISPACK available with the GA tool is replaced with MAGMA eigensolver. We tested the integration for various input molecules like Ozone, Cr_2. Since the input molecules available for testing have relatively small matrix size, we separated the Global Array eigensolver from the NWChem calculation and ran the tests separately from the NWChem execution. This allowed us to experiment with larger matrix sizes in the MAGMA-based eigensolver. Specifically, the test case uses the Global Array tools (which NWChem also calls to calculate the eigenvalues) where the existing EISPACK sequential eigenvalue solver is replaced with the GPU eigenvalue solver from the MAGMA package.

Table 2: **Time comparisons for various DFTFOCK CUDA implementations**

Experiment	Total execution time(Milli s)	Memory allocation time(Milli s)	Total Kernel execution time(Micro s)
A	70	70	87
B	3.8 $*10^3$	73	270
C	76	73	270
D	74	73	87
E	75	73	110

Table 3: **Comparison between CPU and GPU implementation times of DFT in NWChem**

Size of Matrix	Total CPU execution time(s)	CPU+GPU execution time (Shared)(s)	Speed up
10	0.000	4.3954	Nil
100	$1.326*10^{-3}$	7.8526	Nil
1000	15.1234	17.2368	Nil
2000	38.3678	27.8459	1.3
5000	228.398	80.2697	3
10000	876.679	158.9643	5

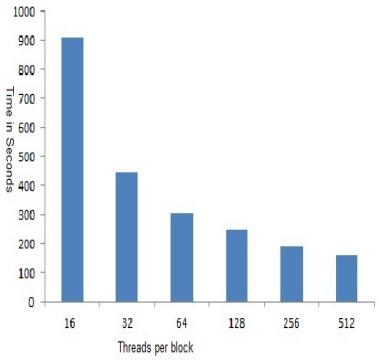


Fig. 5: Timings in DFT calculations using various number of threads in NWChem

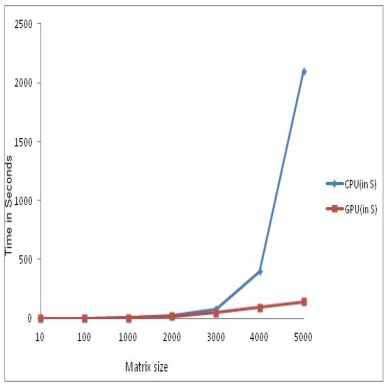


Fig. 6: Speedup in eigenvalues calculation using MAGMA

Table 4 shows the speedup with various matrix sizes. If the matrix size is less than 2000 there is no speed up. So a GPU implementation for the eigenvalues can be effective if the size of the matrix is around 2000. The speedup increases with increase in matrix size.

Figure 6 shows execution time for various matrix sizes on CPU and GPU respectively. The speedup starts to show when the matrix size reaches 2000. To summarize, it is advisable to keep data on the GPU memory, coalesce global memory accesses to reduce latency of data transfer, take advantage of shared memory, and to use hybrid code with double-precision applications.

6. Related work

Several other quantum chemistry applications have been implemented on GPU to exploit the greater level of parallelism. The strategy and optimization techniques for back porting an optimized GPU kernel to a multi-core CPU platform for the application TeraChem - a quantum chemistry code that was developed from the ground up to run on NVIDIA GPUs - is discussed in [13]. For this study, one of TeraChem's largest and most complex GPU kernels is considered. This kernel is used to calculate the electron repulsion integrals involving d-functions. It also investigates which CPU-specific optimizations can be applied to improve performance of the backported kernel. The exploitation of Quantum Monte Carlo algorithms with multiple forms of parallelism and its package simulation code portability to the NVIDIA CUDA GPU platform are discussed in [2]. The restructuring of the CPU algorithms to express additional parallelism, minimize GPU-CPU communication, and efficiently utilize the GPU memory hierarchy is important in porting the CPU code to GPU [2]. The restricted Hartree-Fock method is implemented on a multi-GPU system (a conventional cluster outfitted with GPUs) and its effectiveness is demonstrated [9]. According to [11], GPUs can significantly

Table 4: **Speedup in eigenvalues calculation using MAGMA**

Size of Matrix	Total CPU execution time(s)	GPU execution time(s)	SpeedUp
10	0.00001	0.001300	Nil
100	0.00399	0.399	Nil
1000	2.4650	2.4550	Nil
2000	22.91551	19.17751	1.2
3000	80.09583	50.1824	2
4000	400.1783	95.3697	4.19
5000	2100.7624	143.2457	15

outpace commodity CPUs in the central bottleneck of most quantum chemistry problems. It also explains the method to separate memory bound operations by modifying the algorithm and the memory scheme. It also demonstrates speedups are readily achievable for chemical systems of practical interest, and the inherent high level of parallelism results in complete elimination of inter-block communication. Due to the relative number of single and double precision cores, the best performance for GPU accelerated code is achieved when performing operations at single precision. It also discusses various issues that come with GPU implementations like memory transfer, accuracy, and thread consistency [5].

7. Conclusion and future work

In this work, a GPU integration for two widely used quantum chemistry packages, GAMESS and NWChem, is successfully performed. The exploration of two applications gives insights into GPU needs, and the experiments results demonstrate that there is a trade-off between performance gains and the ease of integration. This work focuses mainly on the experimental exploration of two large code bases: it pinpoints the most time consuming part in the DFT algorithm and links the GPU based eigensolver for NWChem. The intention was to integrate the GPU support without much source code changes. To accomplish this goal, the standalone units are identified, such as the eigensolver in the DFT algorithm. However, in order to fully exploit the available GPU, several software strategies have to be carefully designed and implemented. Currently, the performance bottleneck is in the data transfer between CPU and GPU. Performance tuning and further investigation of adapting GPU to more complex algorithms used in computational chemistry is ongoing.

Acknowledgment

This work was supported in part by Ames Laboratory (Iowa State University) under the contract DE-AC02-07CH11358 with the U.S. Department of Energy, by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231, and by the National Science Foundation grants NSF/OCI – 0749156, 0941434, 0904782, 1047772. The authors are have benefited from many helpful discussions with Professors Theresa L. Windus and Mark S. Gordon and their students at Iowa State University.

References

[1] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[2] K. Esler, J. Kim, L. Shulenburger, and D. Ceperley. Fully accelerating quantum monte carlo simulations of real materials on gpu clusters. *Computing in Science Engineering*, PP(99):1, 2010.

[3] Mark S. Gordon and Michael W. Schmidt. Advances in electronic structure theory: Gamess a decade later. *Theory and Applications of Computational Chemistry:the first forty years, C.E.Dykstra, G.Frenking, K.S.Kim, G.E.Scuseria (Editors)*, 2005.

[4] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:B864–B871, Nov 1964.

[5] Karl Wilkinson Kevin J. Naidoo and Kyle Fernandes. Accelerating scientific computing code in fortran: The quantum chemistry project. *PGI Insider*, oct 2011.

[6] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133–A1138, Nov 1965.

[7] Wen mei W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Elsevier, 2011.

[8] Michael W. Schmidt, Kim K. Baldridge, Jerry A. Boatz, Steven T. Elbert, Mark S. Gordon, Jan H. Jensen, Shiro Koseki, Nikita Matsunaga, Kiet A. Nguyen, Shujun Su, Theresa L. Windus, Michel Dupuis, and John A. Montgomery, Jr. General atomic and molecular electronic structure system. *J. Comput. Chem.*, 14(11):1347–1363, 1993.

[9] Guochun Shi, V. Kindratenko, I. Ufimtsev, and T. Martinez. Direct self-consistent field computations on gpu clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –8, april 2010.

[10] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines — EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science, Editors: G. Goos and J. Hartmanis*. 1976.

[11] I.S. Ufimtsev and T.J. Martinez. Graphical processing units for quantum chemistry. *Computing in Science Engineering*, 10(6):26 –34, nov.-dec. 2008.

[12] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477 – 1489, 2010.

[13] Dong Ye, A. Titov, V. Kindratenko, I. Ufimtsev, and T. Martinez. Porting optimized gpu kernels to a multi-core cpu: Computational quantum chemistry application example. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 72 –75, july 2011.

Power Aware Tactical Computing

Song J Park[1], Dale R Shires[1], Brian J Henz[1], James A Ross[2], David A Richie[3], and Jordan J Ruloff[2]
[1]U.S. Army Research Laboratory, APG, MD
[2]Dynamic Research Corp., Andover, MA
[3]Brown Deer Technology, Forest Hill, MD

Abstract - *Power consumption has become a chief impediment in the advancement of digital computing. To improve performance amid power limitations, accelerators are being applied to everyday systems. In particular, due to the mix of popularity and raw computational power, graphics processing units (GPUs) have extended the applicability of digital computing in a multitude of sectors from ubiquitous smartphones to environmentally responsible supercomputing. Operating within the mobile power constraint, the usefulness of a high performance graphics processor system in a tactical environment is explored in this study. A line-of-sight optimization algorithm serves as a compute-intensive application with characteristics relating to a tactical scenario. Power-efficient hardware options and achievable parallel computing capabilities are analyzed for assisting tactical operations.*

Keywords: GPGPU, line-of-sight, mobile HPC

1 Introduction

Computing power continues to increase as new generations of processors are released to consumers. Following Moore's Law, where the number of transistors double every 18 months, computing elements within processing units have grown exponentially since the birth of digital computing. By common logic, higher transistor count translates into higher performance potential. The TOP500 maintains statistical lists of supercomputers around the world. High performance systems are ranked according to the performance outcome of the Linpack benchmark, which tests the ability to solve a dense system of linear equations. In a nutshell, the benchmark algorithm evaluates the compute speed of double-precision arithmetic on a given system [1]. The TOP500 table is released biannually and includes technical details such as processor type, number of processors, theoretical peak performance, and maximum Linpack performance. Since 2008, detail on power has been included as part of the statistical fields, a hint of the important role power will play in future systems as computing field advances to reach exascale performance.

The U.S. Army Research Laboratory (ARL) Department of Defense Supercomputing Resource Center (DSRC) supports and maintains high performance computing (HPC) resources. The ARL DSRC provides state-of-the-art computational solutions for the DoD research and development community. Among the systems available at ARL is Harold system, which consists of 10,752 cores with the processing capability of 120 trillion floating-point operations per second (TFLOPS). A predecessor, JVN system, decommissioned in 2009, had 2048 cores with the theoretical peak of 14.7 TFLOPS. Shifting the focus to single-precision, the peak processing power for the Harold system is 240 TFLOPS and 29.5 TFLOPS for the JVN system. Given that a single graphics card, Radeon HD 6990, is rated at 5.1 TFLOPS for peak single-precision arithmetic, this means a single video card is equivalent to roughly 1/6 of the JVN's theoretical compute power. In other words, networking six AMD Radeon HD 6990 PCI-E boards can provide a combined peak performance exceeding that of the available 32-bit floating-point power in JVN, albeit single-precision. A graph showing peak performance values for GPUs and their relationship to ARL's previous-generation supercomputers is presented in Figure 1. The peak comparison figure reflects the current state of raw compute abilities offered by common GPU cards and systems, which rival decommissioned yet once TOP500 ranked systems. Moreover, an equivalently powerful system using accelerators built today requires less space, power, and cost to operate. This is the motivation behind leveraging GPU accelerators to enhance power-constrained tactical computing.

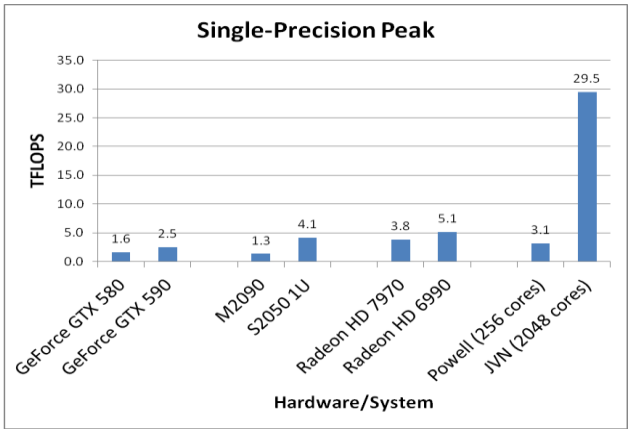


Figure 1. Single-precision comparison of GPU devices with previous-generation supercomputers at ARL

Originally designed to drive displays, GPU cards have become quite powerful in executing single-precision calculations to account for the parallel nature of drawing graphics on a screen. Since 2007, general purpose computing on graphics processors has introduced CUDA and OpenCL languages to support general C-based algorithms to access the underlying parallel hardware. This effectively and seamlessly boosted performance in systems, which most likely had graphics processors installed. In this respect, the GPU approach is an affordable and readily available consumer product with some programming effort since its characteristics are similar to C language. The result of continuing advancement of consumer-grade GPUs has made HPC available to the masses.

2 Influenced by Power

Examples of power influencing design can be observed at multiple levels of computing hierarchy, starting from supercomputing centers to transistor designs. Topics relating to power issues in computing market are summarized in this section where multitudes of options for reducing power consumption are briefly visited.

The cost to power and cool a system is usually one of the limiting factors and is a challenge for hosting large scale HPC systems. The average electrical power consumption for petaflop systems in the TOP500 in November 2010 was 3.3 MW, which translates to $2.89 million per year, assuming $0.10/KWh. As the supercomputing community strives to achieve exascale computing, efficiently managing energy and power will be one of the major challenges. The emphasis on power is evident on the ubiquitous HPC DARPA project soliciting for a prototype of a petaflop-class rack system drawing 57 KW by year 2014 [2].

The shift toward multiple cores in semiconductor giant Intel and AMD is another example of how power and heat have affected central processor architecture design. In order to avoid producing more heat per square centimeter than the surface of the sun, while continuing to improve performance, multicore was introduced as a solution. Yet, the clock frequency did not advance during this period, but rather relied solely on the resource parallelism for performance gain. Additional attention to power is manifested via Intel Sandy Bridge's implementation of on-die power meters that can measure power use on the chip and can dynamically distribute power [3]. Intel's 3D transistor gate is another technique for power reduction. Novel vertical fins of the semiconductor substrate mean more energy efficiency resulting from lower voltage operation and lower leakage [4]. Moving forward, the largest transistor allocation inside an Intel Nehalem die relates to cache. Distributed L1 and L2 cache inside each core and the shared L3 cache clearly dominate the transistor usage. This implies that the majority power is dissipated in the cache circuitry of the CPU. Coupled with the energy required to transfer data from main memory, data movement seems to impact power consumption at a multiple memory hierarchy.

Similarly, features to optimize power consumption are evident in graphics architecture. For example, advanced power management techniques available in Nvidia include multiple levels of clock gating, dynamic voltage scaling, and adaptive clocking. Clock gating shuts off clocks during idle conditions, voltage scaling drops voltage levels during idle states, and adaptive clocking automatically adjusts core and memory frequency to save power. Scientists and engineers at ARL have collaborated with researchers at University of Texas El Paso in looking at selected use of double-precision functions for Nvidia compute resources to reduce energy consumption. In order to analyze accurate measurement of power usage, probing equipments are installed to capture voltage and current signals. Power efficient computing is addressed from a software perspective in [5].

Field-programmable gate arrays (FPGAs) are an option for application-tailored, low-power processing. Altera's Stratix V FPGAs are a low-power option that supports floating-point implementations with a variable-precision DSP architecture. A white paper by Altera's describes that the single-precision multiplier density in Stratix V has increased to 4096, which calculates to floating-point processing rates in excess of 1 teraflops [6]. In terms of the metric flops per watt, this equates to 10 GFLOPS/W. Yet, the programming model and development processes are not as friendly to those familiar with working in C.

In the fourth quarter of 2010, sales of ARM-based smartphones exceeded the sales of personal computers [7]. ARM has historically emphasized the metric of power efficiency due to having evolved in a low-power, battery-operated environment. ARM processors have gained dominance in the segment of the mobile market. Additionally, ARM is looking to enter the server market as the efforts are underway by companies like Calxeda to implement large-scale ARM powered servers. Current Calxeda products offer server nodes operating at 5 W using Cortex-A9 cores [8]. For hosting and servicing web applications, ARM processor's low power consumption offers promising alternative to the traditional server platforms.

3 Hardware for Tactical Computing

Workstation footprint system is a manageable-sized PC augmented with high-end graphics processors. This concept of HPC in a box to aid on-field computations was investigated from two angles. First, the upper limit as to how much computing power can be packed into a workstation

form factor was explored. Secondly, development software and algorithms were analyzed for feasibility and possibilities. Initial attempts at the asymmetric workstation system involved switching heat sinks and fans with custom fitted water blocks for GPU cards to populate the system with seven Radeon HD 4870X2 cards. Since Radeon HD 4870X2 cards contain dual graphics engines, it is equivalent to having 14 GPU devices inside a single workstation. Additionally, liquid cooling allowed for overclocking the reference clocks to achieve extra performance. The target goal was to reach 20 TFLOPS mark for single-precision operations. In regards to software development environment, both CUDA and OpenCL frameworks, which extend the power of the GPUs beyond graphics, were evaluated for parallel computing. CUDA is proprietary and thus specifically supports Nvidia GPU architectures. CUDA was made public in 2007 and was originally based on Open64 C compiler until the recent switch to low level virtual machine (LLVM) for CUDA release 4.1. Nvidia's earlier start has served to acquire initial momentum within the GPGPU community extending to Matlab, Python, and Fortran, to name a few. OpenCL, on the other hand, is vendor agnostic that can target x86 processors, AMD GPUs, and Nvidia GPUs. In both frameworks, a language is provided for writing kernels representing core functions and application programming interfaces (APIs) that are used to manage the platforms. Ongoing translation research continues for compiling OpenCL programs to run on FPGAs and ARM processors.

More recently acquired test bed systems at ARL include a Nvidia GeForce 580 system and an AMD Radeon HD 6970 GPU system. The configuration for the 4U workstation platforms are dual-socket Hex-core Xeon system, 24 GB DDR3 memory, double-width PCIe slots for holding four GPU cards, a solid-state drive, and 1400 W power supply. The goal is for this system to target compute-intensive calculations inside a mobile platform. Typically, an average alternator in a vehicle is rated at 100 A at 12 V, which computes to 1200 W. Thus, the power requirement of the four GPUs workstation seems to be within an achievable range for an automobile operation.

ARM powered devices are explored for integrating with the GPU workstation platforms for a tactical computing scenario. One of the purposes for mobile ARM products would be to serve as an end user interface to display supercomputing calculated results. For instance, iPhone can request ray-tracing calculations to a nearby GPU-enhanced workstation to achieve near real-time computation. To learn the development process and the compute capabilities of an ARM processor, PandaBoard ES was procured for evaluation and its possible role in mobile HPC. PandaBoard ES has the OMAP 4460 processor that is designed by Texas Instruments, which is a combination of the ARM architecture series Cortex-A9 and an Imagination Technologies POWERVR graphics core. Once a Linux operating system is loaded on a PandaBoard, Army applications written in C can be benchmarked. Currently, OpenCL support for ARM is still in its initial stage and a STDCL support is under development. Linux distributions, Angstrom and Ubuntu, were successfully tested on a PandaBoard. A noticeable slow behavior was observed in the GUI mode of Ubuntu. Currently in process of examination is the open source Android operating system that has quickly gained popularity in smartphones and tablets. Although, still relatively immature, Android has a potential to be Microsoft Windows for the next decade.

4 Army-centric Application

An algorithm for ballistic field calculation based on first-hit ray-tracing method was developed using OpenCL. The ray-tracing method allows for the calculation of line-of-sight ballistic threat locations within a specified area. A three-dimensional representation of a small part of town was selected for testing the ballistic hit probability field calculations. The input for the algorithm is a triangle data format describing a three-dimensional surface and layout. For user interaction and displaying of results, functionalities were leveraged and added to the World Wind program. World Wind was developed by NASA and is an open-source interactive world viewer. Figure 2 shows the World Wind interface window with loaded elevation map and the ballistic hit probability field, denoted by red shaded overlay on the surface. In the figure, the end user can insert entities, which are color coded to designate red as a shooter, green as an observation point, and blue as a watcher. A shooter was placed on top of a roof and an observation point was set near the front wall of the building to represent a door of interest. Executing the ballistic threat minimization algorithm generates the optimal locations for a watcher that minimize threat level while maintaining line-of-sight to the observation point.

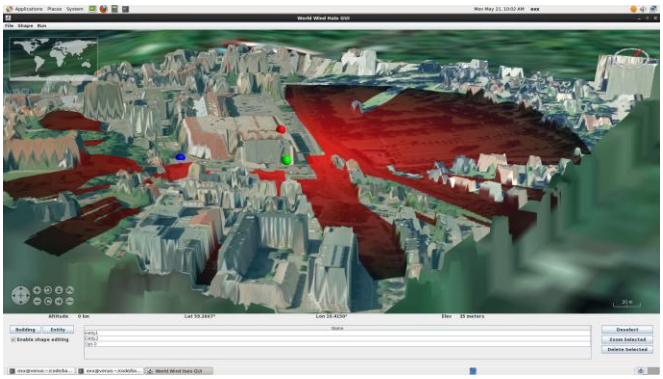


Figure 2. Interface to ballistic threat minimization

The example case consisting of two shooters and two observation points was benchmarked on a single Nvidia GPU machine and an AMD multi-GPU workstation. The Nvidia

system contained Tesla C2050 graphics card and the multi-GPU workstation had four Radeon HD 6970 cards. The code was written to support multi-GPU execution, but to avoid serialization between GPUs; an older version of AMD graphics driver was required for CentOS Linux operating system. Timing results for ballistic threat calculation and placement optimization are presented in Figure 3. Written in OpenCL, the algorithm is portable to x86 processors as well. The calculation on a dual-core Xeon 5160 completed in 25 minutes.

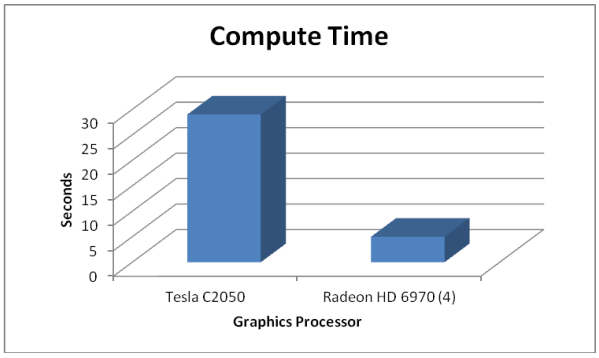


Figure 3. Timing measurements for ballistic threat minimization

5 Conclusion

Armed with mass-marketed processors, Army applications are targeted for enhancement with heterogeneous computing solutions. With the assistance of GPUs, computationally powerful systems can be placed closer to field operations in a small form factor and can handle processing requirements onboard. Furthermore, ARM-based devices with portability and low-power characteristics are naturally appropriate interface in tactical operations. A GPU workstation will be responsible for complex and time-consuming computations of algorithms while an ARM product will provide an intuitive interface and abstraction. Future work for ARM development boards would be to assess and analyze ARM systems in a networked configuration, running cloud computing software.

This research investigated the applicability of asymmetric core computing in a battlefield environment. Across computing disciplines, power related issues play a dominant role in architecture, design, and systems. This work hopes to complement and facilitate the transition of heterogeneous computing to battlefield operating systems.

6 References

[1] TOP500 Supercomputer Sites. Available: http://top500.org

[2] DARPA, Broad Agency Announcement Ubiquitous High Performance Computing Transformational Convergence Technology Office, March 2010.

[3] R. Merritt, "Intel offers first tour of its bridge to heterogeneous computer processors," EE Times. Sep 2010.

[4] Intel Newsroom 22nm 3-D Tri-Gate Transistor Technology. Available: http://newsroom.intel.com/docs/DOC-2032

[5] Ricardo Portillo, Sarala Arunagiri, Patricia J. Teller, Song J. Park, Lam H. Nguyen, Joseph C. Deroba and Dale Shires, "Power versus performance tradeoffs of GPU-accelerated backprojection-based synthetic aperture radar image formation," Proc. SPIE 8060, 2011.

[6] Altera Corporation, "Achieving One TeraFLOPS with 28-nm FPGAs," Sep 2010.

[7] "Flying Robots Designed to Form Emergency Network," Computer, vol. 44, no. 5, pp. 14-16, May 2011.

[8] Calxeda. Available: http://www.calxeda.com

A Parallel Algorithm for Constructing Obstacle-Avoiding Rectilinear Steiner Minimal Trees on Multi-Core Systems

Cheng-Yuan Chang and I-Lun Tseng
Department of Computer Science and Engineering
Yuan Ze University, Taiwan

Abstract - *In the field of integrated circuit physical design automation, the problem of obstacle-avoiding rectilinear Steiner minimal tree (OARSMT) construction is a fundamental problem and has attracted a lot of research attention. In this paper, a parallel algorithm for constructing OARSMTs is proposed. The algorithm is based on maze routing and double front-wave expansion. Experimental results show that our algorithm not only generates very short wires, but also performs efficiently on shared-memory multi-core computer systems. Compared with the sequential execution of our parallel program that has been implemented, the program can achieve 23% speed-up on average while running on a multi-core workstation.*

Keywords: VLSI Physical Design Automation, Routing, Parallel Programming, Multi-Core, Algorithm

1 Introduction

As the semiconductor process technology advances, transistor feature sizes have been shrinking. As a result, an integrated circuit can accommodate many transistors and building blocks. To implement an integrated circuit containing complex functions, a number of building blocks are usually required. Since each building block performs a specific set of functions (e.g., inverter, NAND, NOR, shift register, memory, and ALU), we can combine many building blocks and make proper connections between their terminals in order to build a larger circuit. Routing algorithms, which are capable of making connections between building blocks, play a crucial role, especially when the design is large or complex.

In the field of integrated circuit physical design automation [5, 6], many routing algorithms require the function of constructing *rectilinear Steiner minimal tree*s (RSMTs). That is because most integrated circuit layouts use rectilinear wires. Also, by minimizing the total wire length of an integrated circuit, the signal propagation delay, the power consumption, and the die size can all be reduced. Since an integrated circuit layout can contain building blocks and pre-routed nets, it is desirable to consider regions that cannot be passed through by a net (or wire) as obstacles. Consequently, developing efficient algorithms for solving the *obstacle-avoiding rectilinear Steiner minimal tree* (OARSMT) problem is important and has thus received a lot of research attention [1-4, 7-10].

Although many algorithms have been proposed in order to solve the OARSMT problem, most of these algorithms are sequential, rather than parallel. As inexpensive multi-core processors and computer systems have become widely available, developing parallel algorithms allows us to exploit the computational power from these shared-memory multi-core systems.

This article presents a parallel algorithm for constructing OARSMTs. The algorithm is based on Watanabe's Steiner tree construction algorithm [12], but there are differences between these two approaches. First, our algorithm is suitable for use on a shared-memory multi-core computer system, whereas Watanabe's algorithm is suitable for use on a computer system containing a two-dimensional array processor. Specifically, an array processor can have as many as tens of thousands of processing elements, while a shared-memory multi-core computer may have less than ten processor cores. Second, our algorithm adopts two parallelized procedures, namely PARALLEL-CONNECT() and PARALLEL-CLEANUP(), and these two procedures can efficiently reduce the program execution time. The parallel algorithm proposed in this article has been implemented in C++ with the use of OpenMP [13]. Experimental results show that the implemented parallel program performs efficiently on a shared-memory multi-core workstation.

The remainder of this article is organized as follows. In Section 2, we formulate the OARSMT problem. Our parallel algorithm for constructing OARSMTs is presented in Section 3. Experimental results are given in Section 4. Finally, conclusions are drawn in Section 5.

2 Problem Formulation

In the *xy* plane, let $T = \{t_1, t_2, \ldots, t_m\}$ be a set of m terminals which are to be connected by a net (or wire), and let $O = \{o_1, o_2, \ldots, o_n\}$ be a set of n rectilinear obstacles. A terminal can exist on the boundary or on a corner of an obstacle, but cannot exist within an obstacle. In Fig. 1(a), for example, t_2 is a valid terminal since it is on the boundary of an obstacle, whereas t_3 is an invalid terminal since it is located

This research was supported in part by the National Science Council of Taiwan under grants NSC-98-2221-E-155-053 and NSC-99-2221-E-155-088.

within the obstacle. Invalid terminals cannot appear in an instance of the OARSMT problem. For the figures in this article, obstacles are denoted by light-gray regions.

With regard to physical positions of line segments of a wire, a line segment can overlap with the boundary of an obstacle as shown in Fig. 1 (b), but cannot pass through the interior of an obstacle as shown in Fig. 1 (c). Since a wire is composed of rectilinear line segments, the *total wire length* of a routing result is computed as the sum of the lengths of all line segments. Fig. 1 (d) illustrates a valid routing result and its total wire length is 8. The OARSMT problem is defined below.

Definition 1 (The OARSMT Problem):

Given a set T of terminals and a set O of rectilinear obstacles in the xy plane, we are requested to construct a net (or wire) which consists of rectilinear line segments and connects all the terminals in T, while possibly through some Steiner points. In addition, line segments of the net cannot pass through the interior of each obstacle in O, and the total wire length of the routing result must be minimal.

3 The Algorithm

Our approach for constructing an OARSMT is based on Watanabe's algorithm [12], which mainly consists of two repetitive phases (known as *double front-wave expansion*). The first phase is called the *first front-wave propagation*; a set of start points P_S is selected, and then we propagate from these points in order to find another terminal $t_t \in T$, which is the closest to P_S. During the propagation process, in addition, the propagation wave cannot pass through the interior of each obstacle. Also, the shortest distance between a point in P_S and each visited grid point must be recorded; the numbers representing these distances are called *distance numbers*.

The second phase of Watanabe's algorithm is named the *second front-wave propagation*; another propagation wave starts from the terminal t_t to reach either a point or points in P_S. An overlapping region between the first and the second phases can thus be found. Moreover, Steiner points can also be determined. Therefore, an OARSMT can be obtained after connecting all the Steiner points and terminals.

Our parallel algorithm for constructing an OARSMT from given sets of T and O is described in Fig. 2. First, a start terminal (t_s) must be selected from T. Although an arbitrary terminal can be selected from T as the start terminal, our implementation of the algorithm selects the terminal which has the least value in terms of the sum of x and y coordinates. The set V is used for storing all the terminals that have been visited during double front-wave expansion. Since the algorithm starts by propagating from the start terminal t_s, it is assigned to V in line 2. The set S, which can be implemented as a task queue, is used for storing grid points from which the next iteration of double front-wave expansion will start; these grid points may include terminals and/or Steiner points.

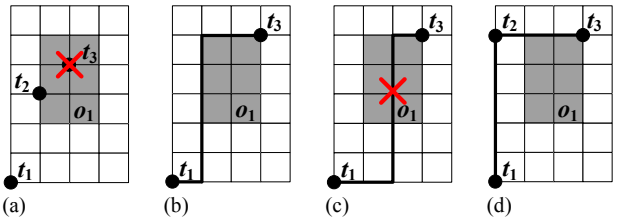


Figure 1. Valid and invalid OARSMT examples and routing results.

Algorithm PARALLEL-OARSMT-CONSTRUCTION(T, O)

Input: T /* all the terminals which are to be connected */
O /* all the rectilinear obstacles */

Output: An OARSMT which connects all the terminals in T using only horizontal and/or vertical line segments while these line segments cannot pass through the interior of each obstacle.

1. $t_s \leftarrow$ select a start terminal from set T
2. $V \leftarrow t_s$
3. $S \leftarrow t_s$
4. **while** ($|V| < |T|$) {
5. $t_i \leftarrow$ PROPAGATE_1ST(S)
6. $p \leftarrow$ PROPAGATE_2ND(t_i)
7. $R \leftarrow$ the intersection region between the first and the second front-wave propagation phases.
8. Insert t_i into V.
9. **if** ($p \neq$ NULL **AND** $p \notin T$)
10. PARALLEL-CONNECT()
11. PARALLEL-CLEANUP()
12. $S \leftarrow \phi$
13. $S \leftarrow$ all the grid points from which the first front-wave propagation in the next iteration will start.
14. }
15. FINAL-CONNECT()
16. **return** the completed OARSMT

Figure 2. The proposed parallel algorithm for OARSMT construction.

The **while** loop in line 4 checks whether all the terminals have been visited or not. Lines 5–13 will be executed if at least one terminal in T has not been visited. The first front-wave propagation phase is implemented in the procedure PROPAGATE_1ST(); the procedure returns a terminal (t_i) which is the terminal found in the phase and is the closest terminal to grid points in S. Note that distance numbers increase as the propagation proceeds during this phase.

The second front-wave propagation phase is implemented in the procedure PROPAGATE_2ND(). The procedure returns p, which is either a newly found Steiner point or NULL. Note that distance numbers decrease as the propagation proceeds during this phase. In line 7, a region R, which is the intersection of regions between the first and the second propagation phases, can be obtained. In line 8, the terminal that has just been discovered during the first front-wave propagation phase is inserted into V.

In lines 9–10, if Steiner point p exists and it is not an element in T, the wire connections between p and terminals in a region which was formed previously will be made. We have found that the connection process can be parallelized; it has been implemented in the procedure PARALLEL-CONNECT(). The procedure can efficiently speed up the connection process and will be described in detail in subsection 3.1.

Since distance numbers are recorded during double front-wave expansion, parts of these numbers must be cleared in preparation for the next double-wave expansion iteration. Because the process of clearing distance numbers does not have any data dependency, the process can be parallelized; the parallelized process has been implemented in the procedure PARALLEL-CLEANUP() and will be described in detail in subsection 3.2.

In lines 12–14 of our parallel algorithm, the set S is reset and then its content is updated in preparation for the next iteration of double-wave expansion. After all iterations of double-wave expansion have been completed, there can still exist terminals that remain unconnected. The procedure FINAL-CONNECT() will complete the routing for these unconnected terminals.

Fig. 3 illustrates an example of constructing an OARSMT by using our algorithm. As shown in Fig. 3(a), an instance of the OARSMT problem, which is to be solved, contains four terminals and three obstacles. If t_1 is the start terminal, the first front-wave propagation phase will generate distance numbers, as shown in Fig. 3(b), after t_2 is reached. The second front-wave propagation phase, which starts from t_2 and then reaches t_1, will generate the result as illustrated in Fig. 3(c). Note that the second propagation phase will only visit grid points that have been visited previously in the first propagation phase. Therefore, the overlapping region (region A) between the first and the second front-wave propagation phases can be obtained, as shown in Fig. 3(d). After the first iteration of the **while** loop finishes, set S contains all the grid points which are on the boundary of region A; these points include terminals t_1 and t_2. During the second iteration of the **while** loop, terminal t_4 will be visited and then region B will be generated. A Steiner point, which is at the intersection between regions A and B, can be obtained; the point is named p_1 in Fig. 3(e). The procedure PARALLEL-CONNECT() can then be invoked to connect p_1 and t_1 as well as to connect p_1 and t_2, as illustrated in Fig. 3(f). After that, the third iteration of the **while** loop will visit t_3, and region C in Fig. 3(g) will be formed. Next, Steiner point p_2 will be obtained, and it can then be connected to p_1 and t_4. Finally, the unconnected terminals (i.e., t_3 only in this example) can be connected after invoking the procedure FINAL-CONNECT(). The final routing result is shown in Fig. 3(h).

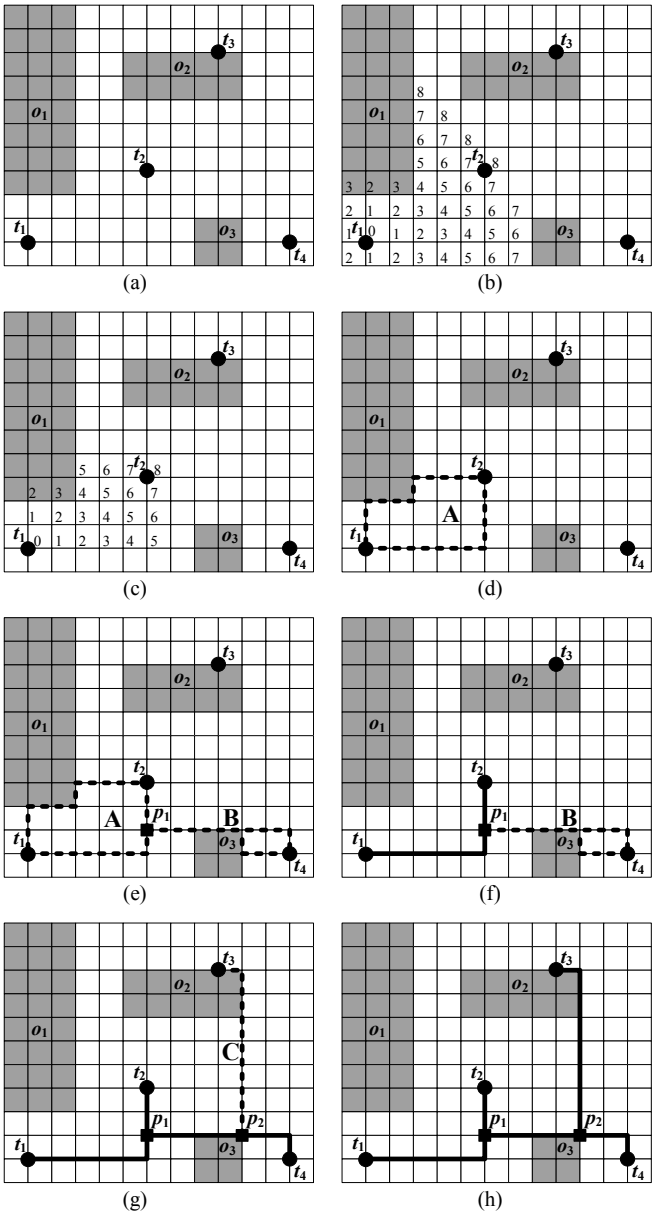


Figure 3. An example of constructing an OARSMT by using our algorithm.

3.1 Parallel-Connect

In Watanabe's PAR-1 algorithm [12], a parameter named *expansion distance* (denoted by D_{ex}) is used to control the quality of routing during front-wave propagation phases. When D_{ex}=1, the PAR-1 algorithm is equivalent to Lee's algorithm [14], and a wire whose length is the shortest between two given points can be obtained. On the other hand, when $D_{ex}=\infty$, a wire which has the minimum number of bends can be generated, although the wire length may not be the shortest.

The procedure PARALLEL-CONNECT() is used to connect a Steiner point with other points, including terminals and/or other Steiner points, within a region. The technique of front-wave propagation is also used in the procedure. However, the procedure is parallelized and parameter D_{ex} is set to infinity. Therefore, connections will be made within the region and have the minimum number of bends. As a result, final routing results generated by our algorithm can have minimal total wire lengths and minimal number of bends.

An example for illustrating detailed steps of PARALLEL-CONNECT() is shown in Fig. 4. In this example, we assume that Steiner point p_1 has been found, and we need to connect p_1 to t_1 as well as connect p_1 to t_2, as shown in Fig. 4(a). Also, connections must be made within region A. After the front-wave propagation phase, which starts from p_1, with D_{ex} set to ∞ is finished, distance numbers for visited grid points are shown in Fig. 4(b). In the backtracking phase, terminal t_1 must be connected to p_1 and terminal t_2 must also be connected to

p_1. Since t_1 and t_2 must be located on different corners of region A and p_1 must be on the boundary of region A, the two connection processes are independent and can thus be parallelized. Therefore, the connection between t_1 and p_1 can be made by one thread while the connection between t_2 and p_1 can be made by another thread. Because there are two possible routes for the connection between t_1 and p_1, one of the two possible connection results, shown in Fig. 4(c) and Fig. 4(d), will be generated.

After all the terminals have been visited during iterations of double front-wave expansion, there may still exist terminals that have not been connected. These terminals will be connected via procedure FINAL-CONNECT().

3.2 Parallel-CleanUp

After each iteration of the first and the second front-wave propagation phases, an intersection region R can be obtained, as mentioned previously. However, distance numbers for grid points which are not located in the intersection region must be reset to zero before the next iteration of double front-wave expansion starts. The process of resetting distance numbers of grid points is named the *clean-up* process and can be parallelized. Fig. 5(a) illustrates an example where region A has been generated. After the execution of the clean-up process, distance numbers for grid points outside region A will be reset, as shown in Fig. 5(b).

Another occasion that requires the invocation of the clean-up process is when wire connections within a region have been made, and the region must be removed in preparation for the next iteration of double front-wave expansion. For example, in Fig. 5 (c), region A has been generated, followed by the generation of region B. Also, Steiner point p_1 has been obtained. After terminals t_1 and t_2 have been connected to p_1, therefore, distance numbers for grid points after performing double front-wave expansion as well as distance numbers for grid points inside region A must be reset. However, distance numbers for grid points inside regions which have not been removed must be kept intact. The result after executing the clean-up process is illustrated in Fig. 5(d).

Since the two occasions for performing the clean-up process can be merged, we implemented the clean-up process in procedure PARALLEL-CLEANUP(). The procedure can be parallelized because there is no data dependency while resetting distance numbers for individual grid points.

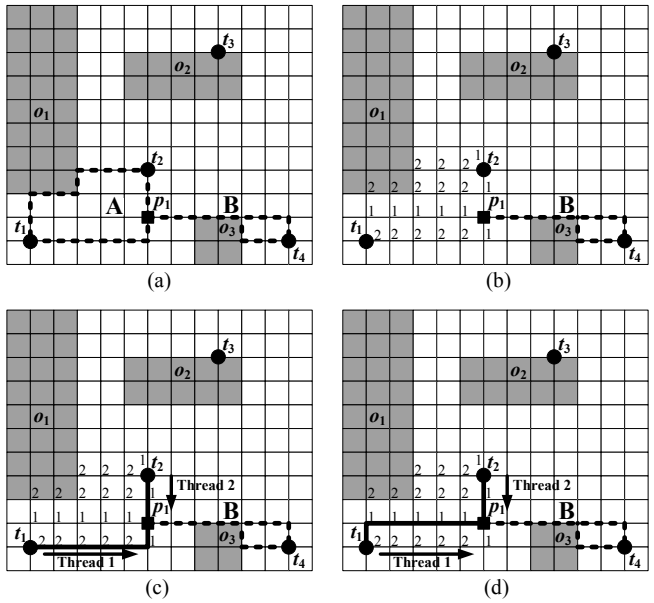


Figure 4. An example for illustrating detailed steps of PARALLEL-CONNECT().

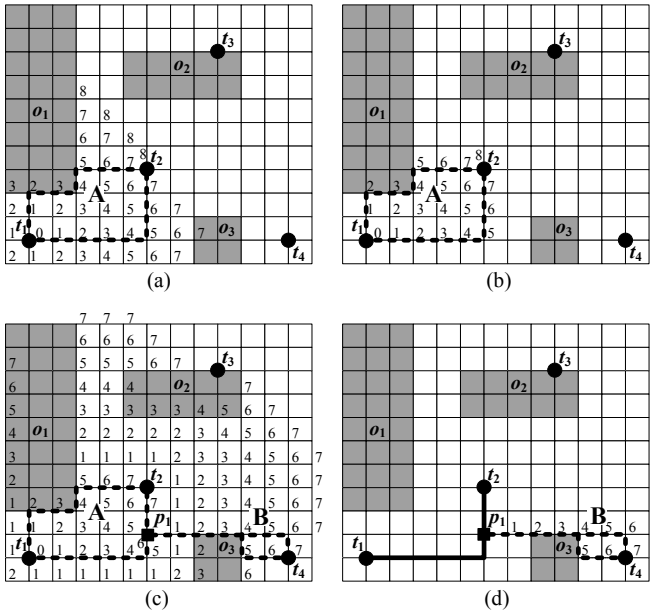


Figure 5. Examples for illustrating functionalities of procedure PARALLEL-CLEANUP().

4 Experimental Results

The algorithm presented in Fig. 2 has been implemented in C++ with the use of OpenMP [13]. Experiments were carried out on a Linux workstation containing two Quad-Core Xeon E5520 2.26GHz processors and 32GB of RAM. Table I shows statistics information for each of 21 benchmark testcases; the information includes the number of terminals (denoted by #Pin) and the number of obstacles (denoted by #Obs). Among these benchmark testcases, five were industrial testcases (ind1 to ind5) from Synopsys, eleven testcases (rc01 to rc11) were used in [7, 8], and five randomly-generated testcases (rt01 to rt05) were used in [1, 8]. Table I also shows the execution times of our program for solving these benchmark testcases. On average, 23% speed-up can be achieved while executing the parallel program using 4 threads for solving these benchmark testcases.

Table II presents comparisons of total wire lengths for routing results generated by different OARSMT algorithms; the testcases listed in the table are the same as the ones listed in Table I. Additionally, Table III shows comparisons of total wire lengths for routing results generated by different

TABLE II. Comparisons of Total Wire Lengths for Different OARSMT Algorithms

Testcase	Lin [1]	Lin [3]	Long [2]	Li [4]	Liu [9]	Liu [10]	Ajwani [7]	Our Program
ind1	632	632	639	619	626	604	604	619
ind2	9600	9600	10000	9500	9700	9600	9500	9500
ind3	613	613	623	600	600	600	600	600
ind4	1121	1121	1126	1096	1095	1092	1129	1100
ind5	1364	1364	1379	1360	1364	1353	1364	1362
rc01	26900	26900	27540	25980	26740	25980	25980	25980
rc02	42210	42210	41930	42010	42070	41350	42110	42280
rc03	55750	55750	54180	54390	54550	54360	56030	54500
rc04	60350	60350	59050	59740	59390	59530	59720	59460
rc05	76330	76330	74630	74650	75440	74720	75000	74910
rc06	83365	83365	86381	81607	81903	81290	81229	81551
rc07	113260	113260	117093	111542	111752	110991	110764	110899
rc08	118747	118747	122306	115931	118349	115516	116047	116272
rc09	116168	116168	119308	113460	114928	113254	115593	113581
rc10	170690	170690	167978	167620	167540	166971	168280	167130
rc11	236615	236615	232381	235283	234097	234875	234416	234748
rt1	2267	2267	2362	2231	2259	2193	2191	2255
rt2	48441	48441	52218	47297	48684	46965	48156	46976
rt3	8368	8368	8645	8187	8347	8136	8282	8168
rt4	10306	10306	10580	9914	10221	9832	10330	9923
rt5	53993	53993	55286	52473	53745	52318	54634	52422
Normalized	1.0188	1.0188	1.0258	1.0010	1.0075	0.9977	1.0063	1

OARSMT algorithms when all the obstacles in each of the benchmark testcases have been removed. As can be seen in both tables, our approach is very competitive in terms of minimizing total wire lengths.

5 Conclusion

We proposed a parallel algorithm which is capable of constructing obstacle-avoiding rectilinear Steiner minimal trees (OARSMTs) in the gridded *xy* plane. The algorithm is based on maze routing and double front-wave expansion. Although our approach does not use refinement techniques, experimental results have shown that it is very competitive in terms of minimizing total wire lengths. Moreover, 23% speed-up can be achieved on average while executing the parallel program using 4 threads on a multi-core computer system.

TABLE I. EXECUTION TIMES OF OUR PROGRAM FOR SOLVING BENCHMARK TESTCASES

Testcase	#Pin	#Obs	Sequential (sec.)	Parallel - 4 threads (sec.)	Speed-Up (%)
ind1	10	32	0.051559	0.030271	41.29
ind2	10	43	1.00103	0.690316	31.04
ind3	10	50	0.028653	0.022504	21.46
ind4	25	79	0.084594	0.047688	43.63
ind5	33	71	0.081815	0.060735	25.77
rc01	10	10	15.3816	12.4315	19.18
rc02	30	10	31.2229	20.8785	33.13
rc03	50	10	65.5559	49.2805	24.83
rc04	70	10	75.34	51.399	31.78
rc05	100	10	152.075	115.974	23.74
rc06	100	500	158.355	128.996	18.54
rc07	200	500	318.218	261.375	17.86
rc08	200	800	312.269	252.955	18.99
rc09	200	1000	273.089	206.761	24.29
rc10	500	100	880.173	741.824	15.72
rc11	1000	100	2056.31	1866.28	9.24
rt01	10	500	0.158169	0.12368	21.80
rt02	50	500	58.0274	46.7059	19.51
rt03	100	500	1.52453	1.32847	12.86
rt04	100	1000	1.47087	1.22571	16.67
rt05	200	2000	47.4726	37.1076	21.83
Average	–	–	–	–	23.48

References

[1] Chung-Wei Lin, Szu-Yu Chen, Chi-Feng Li, Yao-Wen Chang, and Chia-Lin Yang, "Efficient Obstacle-Avoiding Rectilinear Steiner Tree Construction," In *Proceedings of International Symposium on Physical Design*, pp. 127-134, 2007.

[2] Jieyi Long, Hai Zhou, and S. O. Memik, "EBOARST: An Efficient Edge-Based Obstacle-Avoiding Rectilinear Steiner Tree Construction Algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, no. 12, pp. 2169-2182, 2008.

[3] Chung-Wei Lin, Szu-Yu Chen, Chi-Feng Li, Yao-Wen Chang, and Chia-Lin Yang, "Obstacle-Avoiding Rectilinear Steiner Tree Construction Based on Spanning Graphs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, no. 4, pp. 643-653, 2008.

[4] Liang Li and Evangeline F.Y. Young, "Obstacle-Avoiding Rectilinear Steiner Tree Construction," In *Proceedings of International Conference on Computer-Aided Design*, pp. 523-528, 2008.

TABLE III. Comparisons of Total Wire lengths for Different OARSMT Algorithms When Testcases Do Not Contain Obstacles

Testcase	#Pin	#Obs	Long [2]	Li [4]	Chu [11]	Ajwani [7]	Our Program
ind1	10	0	614	619	604	604	619
ind2	10	0	9100	9100	9100	9100	9100
ind3	10	0	590	590	587	587	590
ind4	25	0	1092	1092	1102	1102	1092
ind5	33	0	1314	1304	1307	1307	1304
rc01	10	0	25290	25290	25290	25290	25290
rc02	30	0	40100	40630	39920	39920	40400
rc03	50	0	52560	52440	53400	53050	52280
rc04	70	0	55850	55720	57020	55380	55310
rc05	100	0	72820	71820	73370	72170	71860
rc06	100	0	77886	78068	80057	77633	77492
rc07	200	0	106591	107236	109232	106581	107078
rc08	200	0	109625	109059	112787	108928	108866
rc09	200	0	109105	108101	112460	108106	107897
rc10	500	0	164940	164450	170270	164130	165190
rc11	1000	0	233743	235284	245325	233647	234021
rt1	10	0	1817	1817	1817	1817	1817
rt2	50	0	44930	46109	45291	44416	46028
rt3	100	0	7677	7777	7811	7749	7719
rt4	100	0	7792	7826	7826	7792	7775
rt5	200	0	43335	43586	44809	43026	43383
Normalized	-	-	1.00142	1.00241	1.02942	0.99762	1

[5] Naveed A. Sherwani, *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers, 1999.

[6] Sabih H. Gerez, *Algorithms for VLSI Design Automation*, Wiley, 1998.

[7] Gaurav Ajwani, Chris Chu, and Mak Wai-Kei, "FOARS: FLUTE Based Obstacle-Avoiding Rectilinear Steiner Tree Construction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 30, no. 2, pp. 194-204, 2011.

[8] Tao Huang and Evangeline F.Y. Young, "Obstacle-Avoiding Rectilinear Steiner Minimum Tree construction: An Optimal Approach," In *Proceedings of International Conference on Computer-Aided Design*, pp. 610-613, 2010.

[9] Chih-Hung Liu, Shih-Yi Yuan, Sy-Yen Kuo, and Yao-Hsin Chou, "An O(n log n) Path-Based Obstacle-Avoiding Algorithm for Rectilinear Steiner Tree Construction," In *Proceedings of Design Automation Conference*, pp. 314-319, 2009.

[10] Chih-Hung Liu, Shih-Yi Yuan, Sy-Yen Kuo, and Jung-Hung Weng, "Obstacle-Avoiding Rectilinear Steiner Tree Construction Based on Steiner Point Selection," In *Proceedings of International Conference on Computer-Aided Design*, pp. 26-32, 2009.

[11] Chris Chu and Yiu-Chung Wong, "FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, no. 1, pp. 70-83, 2008.

[12] Takumi Watanabe, Hitoshi Kitazawa, and Yoshi Sugiyama, "A Parallel Adaptable Routing Algorithm and Its Implementation on a Two-Dimensional Array Processor," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 6, no. 2, pp. 241-250, 1987.

[13] Barbara Chapman, Gabriele Jost, and Ruud van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, The MIT Press, 2008.

[14] C. Y. Lee, "An Algorithm for Path Connections and Its Applications," *IRE Transactions on Electronic Computers,* vol. EC-10, no. 3, pp. 346-365, September, 1961.

Bacon: A GPU Programming System With Just in Time Specialization

Nat Tuck
University of Massachusetts Lowell
1 University Ave, Lowell MA 01854
Email: ntuck@cs.uml.edu

***Abstract*—This paper describes Bacon, a data-parallel programming system targeting OpenCL-compatible graphics processors. This system is built upon the existing OpenCL standard in order to make it easier for programmers to write high performance kernels for GPU accelerated applications. The OpenCL C syntax is extended into a new language, Bacon C, intended to make development significantly more convenient and enabling pre-optimizations based on just-in-time specialization as this code is compiled via OpenCL at runtime.**

Benchmarks are provided for matrix multiplication comparing two Bacon implementations to similar OpenCL implementations. Speedups are demonstrated both for naive implementations and when comparing a Bacon implementation of generalized block decomposed matrix multiplication to a hand-vectorized OpenCL kernel. This latter result demonstrates the benefit of the total loop unrolling enabled by just-in-time specialization.

I. Introduction

The use of Graphics Processing Units (GPUs) for general purpose parallel computing has become increasingly feasible over the last few years. In response to the platform specific programming solutions from NVIDIA and Microsoft, Apple developed the OpenCL standard as an open and cross platform programming interface to this hardware. This standard has since been implemented by a number of major vendors including AMD, NVIDIA, Intel, and IBM.

The OpenCL standard[1] consists of two major pieces. First, it defines a programming language called OpenCL C for writing compute kernels to run on parallel hardware. Second, it defines runtime APIs for C and C++ that allow these kernels to be compiled, loaded, and executed from programs running on a host machine.

OpenCL C uses the syntax of C99 and provides a set of built in data types and functions that expose the numeric computation capabilities common to modern GPU devices. The specification explicitly disallows the use of various C99 functionality that is not supported by GPU hardware, including function pointers, recursion, and any sort of dynamic memory allocation or array sizing.

Rather than providing a stand-alone program to compile OpenCL C kernels, the OpenCL C and C++ APIs give developers the pieces necessary to build a compiler into their host program. This allows OpenCL programs to be portable across different hardware archetectures by delaying compilation until runtime when the target GPU device is known. Unfortunately, it also requires each developer to write quite a bit of code to load the kernel source code and perform various other bookkeeping activities.

This paper introduces Bacon, a parallel programming system built on top of OpenCL. Bacon provides two main benefits to developers over the direct use of OpenCL. First, it provides a better user experience through syntax extensions and automatic generation of interface code. Second, it uses an optimization technique called just-in-time specialization which can speed up kernel execution significantly.

Just-in-time specialization operates by delaying compilation of a Bacon kernel until after it is called with a concrete set of argument values. When a Bacon kernel is written, some integer arguments can be marked as specialization variables. When a kernel is first called with a given set of values for those arguments a specialized version of that kernel is generated with the values of those arguments held constant. Just in time specialization enables several optimizations and capabilities that could not be provided otherwise.

These improvements are evaluated using matrix multiplication as a well-known test case. Both simpler code and improved performance are demonstrated.

II. Related Work

A. Partial Evaluation

Specialization, also known as partial evaluation, has been shown to be a very effective optimization by projects such as Tempo[2], which does general partial evaluation of C code at compile time. Just in time specialization was popularized by its use in typing dynamic languages, as shown in the Self language[3]. Just in time specialization on values has been used in Prolog systems by Bolz[4].

B. OpenCL Libraries and Front-Ends

Bacon is not the first attempt to provide an improved programmer experience for OpenCL. Rick Webber has developed an improved C++ API called clUtil[5]. Bindings for other languages, like the JOCL[6] binding for Java, provide APIs at a variety of levels of abstraction.

Another very interesting approach allows the programmer to describe the computation to be executed on the GPU in the same language as the rest of the program. An OpenCL kernel is then generated automatically. This has the potential benefit of increased expressive power and programmer familiarity at the cost of increased complexity and necessarily leaky

```
kernel
Array2D<float> // (1) return values
mat_mul(Array2D<float> aa, Array2D<float> bb)
{
 SETUP: // (2) in-kernel setup section
    global Array2D<float> cc[aa.rows, bb.cols]; // (3) parameterized types

 BODY:
    @range [cc.rows, cc.cols]; // (4) parallel range declaration

    float sum = 0.0;
    assert(aa.cols == bb.rows, // (5) error checking and handling
        "Matrices must have compatible dimensions.");
    for (int kk = 0; kk < aa.cols; ++kk) {
        sum += aa[$row, kk] * bb[kk, $col]; // (6) multi-dimensional arrays
    }
    cc[$row, $col] = sum; // (7) special index variables
    return cc; // (8) return variable declared and selected in-kernel
}
```

Listing 1: Naive Matrix Multiplication in Bacon C

abstraction when the host language contains features that can't be cleanly transformed into GPU code. Examples of this approach include CLyther[7] for Python and ScalaCL[8] for the functional language Scala.

III. The Bacon C Language

Bacon C is based on OpenCL C with extensions for improved usability and to enable the automatic generation of C++ wrapper code. Sample Bacon C kernels that perform matrix multiplication and demonstrate nine of the new features of the language are shown in Listing 1 and Listing 2.

Bacon preserves the OpenCL single program multiple data (SPMD) computation model. A kernel is executed in parallel over a 1D, 2D, or 3D range. Each executing instance of the kernel can query its position in that range to determine which part of the work it is responsible for performing. For example, the kernel in Listing 1 will be executed in parallel once for each element in the output matrix.

Each kernel is separated into SETUP and BODY sections. The SETUP section ((2) in the Listing 1) is for code that will run serially on the host processor while the BODY section contains the code to be executed in parallel. In practice, the SETUP section is primarily used to declare output arrays that can be returned to the host application and to compute the sizes of these arrays.

Unlike OpenCL C kernels, Bacon C kernels can return values ((1) in Listing 1). Any variable of a simple type can be returned, as can any array passed as an argument or declared in the SETUP section. The return statement occurs in the kernel BODY ((8) in Listing 1) and can selected conditionally (e.g. by an `if` statement), but the behavior if different parallel instances of the kernel try to return different values is undefined.

Each BODY includes an `@range` declaration ((4) in Listing 1) that specifies the range it will be executed over in parallel. Within the BODY, the current position in that range is held in special Bacon-specific variables named `$col`, `$row`, and `$dep` for the first, second, and third dimension respectively ((7) in Listing 1). The range is formatted like an array declaration, so a BODY with `@range[4]` will be executed 4 times in parallel with `$col` having the values 0, 1, 2, and 3. Since this is only a 1D range, the values of `$row` and `$dep` will both be zero.

Bacon provides parameterized types ((3) in Listing 1) for 1D, 2D, and 3D arrays using C++-style angle bracket syntax. Both declarations and element access use a comma separated list of numbers in square brackets ((6) in Listing 1). The dimensions of these arrays can be accessed using struct-style dot notation. For example, a three by three by three array of integers called "cube" could be allocated with `int cube[3,3,3];` and the width of that array could be accessed with `cube.cols`.

Additional error handling is provided through the `assert` ((5) in Listing 1) and `fail` keywords which will stop kernel execution and raise exceptions in the host process if triggered. A `fail` is triggered if execution in any thread reaches that statement, while an `assert` is only triggered if its associated condition is false.

Each Bacon kernel has a set of specialization variables. These fall into two categories. First, the dimensions of any arrays passed as arguments to a kernel are always specialization variables. Second, additional specialization variables can be specified explicitly by declaring arguments using the `const` qualifier (like the `blksz` argument in Listing 2). Whenever a kernel is called with a new set of values for its specialization variables a specialized version of that kernel is generated and

Sequence	Traditional (e.g. FORTRAN Kernel)	OpenCL Kernel	Bacon Kernel
1.	Developer writes host application (C++) and kernel (e.g. FORTRAN).	Developer writes host application (C++) and kernel (OpenCL C).	Developer writes host application (C++) and kernel (Bacon C).
2.	—	Developer writes wrapper code (C++) to load and run kernel.	Bacon compiler generates wrapper code (C++) to load and run kernel.
3.	Host application and kernel are compiled.	Host application and wrapper code are compiled.	Host application and wrapper code are compiled.
4.	Host application is run.	Host application is run.	Host application is run.
5.	—	—	Input data is read.
6.	—	—	Bacon library generates specialized OpenCL kernel.
7.	—	Target GPU is identified.	Target GPU is identified.
8.	—	Kernel is compiled for target GPU.	Kernel is compiled for target GPU.
9.	Input data is read.	Input data is read.	—
10.	Kernel is executed on input data.	Kernel is executed on input data.	Kernel is executed on input data.

TABLE I: Lifecycle of Bacon and OpenCL Kernels

executed. Specialized kernels are cached for future calls with the same set of specialization values.

This specialization, in addition to providing performance benefits, allows variable sized arrays in thread-private memory as long as the array size depends only on `const` variables and array dimensions. Since OpenCL does not allow any form of in-kernel dynamic memory allocation, this makes it possible for users to write kernels that would have been difficult to write using OpenCL directly. The blocked matrix multiply kernel in Listing 2 gives an example of this feature.

IV. Kernel Lifecycle

The basic technique of separating out high performance "kernels" from an application and implementing them in a separate language has been used in software development for decades. This even occurs for purely sequential programming. For example, an application written primarily in C++ may have high performance routines written in hand optimized assembly code or FORTRAN. Traditionally, the host application and kernel code are compiled into separate modules and then linked together before execution. This basic sequence is shown in the first column of Table I.

Compiling everything before execution, or ahead-of-time (AOT) compilation, has one major downside: the target hardware is set when the application or module is compiled. Just-in-time (JIT) compilation avoids this problem by delaying compilation until the program is run on a specific machine, allowing the target hardware to be detected dynamically at runtime. OpenCL uses JIT techniques to allow the portability of kernels across the variety of compute-capable GPUs and other parallel acceleration hardware that provide support for the standard. The JIT OpenCL kernel lifecycle is shown in the second column of Table I.

Bacon takes delayed compilation one step further, waiting to compile a kernel until it is actually called and the characteristics of the arguments can be examined. This allows just-in-time specialization to be performed, as shown in the third column of the table.

V. Implementation

The Bacon system consists of two pieces: the Bacon compiler and the Bacon runtime library. The compiler runs at application compile time and parses the Bacon C source, generating a C++ wrapper and a serialized abstract syntax tree (AST). The Bacon runtime library is called from the generated wrapper as the host application is running to load the AST, generate specialized OpenCL C code when a kernel is called, and run that code on the GPU using the OpenCL runtime.

The system is built using Perl and C++. The Bacon compiler parses the source code using Parse::Yapp[9], a yacc-compatible parser generator for Perl. This constructs the abstract syntax tree as a Perl data structure. The C++ wrapper is then generated by traversing this tree.

The generated C++ wrapper provides a C++ function with the kernel's type signature that can be called from the user's application. When this functions is called, the Bacon runtime library loads the AST and traverses it to generate the specialized OpenCL code. Optimizations are performed at code generation time directly from the AST without the use of a traditional low level intermediate representation.

The two optimizations that are performed by the Bacon runtime library are constant propagation and loop unrolling. Constant propagation calculates the values of all the variables that have been marked as `const` by the programmer. If the value of any of these variables cannot be computed from the specialized arguments to the kernel, the Bacon runtime library will throw an exception. This information is used to construct a symbol table, and references to these variables are replaced with their constant integer values in the generated OpenCL code.

Loop unrolling is performed on any loops for which the iteration count and range can be determined after constant propagation. Short loops are fully unrolled. In this case, no loop is passed to the compiler at all. An example of this is shown at (10) in Listing 2. Longer loops are unrolled by some factor that evenly divides the iteration count.

This specialized and optimized OpenCL C code is then passed to the OpenCL compiler provided with the vendor SDK which will perform further optimizations on the generated

```
kernel
Array2D<float>
blocked_mat_mul(Array2D<float> aa, Array2D<float> bb, const uint blksz)
{
 SETUP:
  global Array2D<float> cc[aa.rows, bb.cols];

 BODY:
  @range [cc.rows / blksz, cc.cols / blksz];

  // (9) private variable-sized array
  private Array2D<float> sum[blksz, blksz];
  int ii, jj, kk, gg;

  for (ii = 0; ii < blksz; ++ii) {
    for (jj = 0; jj < blksz; ++jj) {
      sum[ii, jj] = 0.0;
    }
  }

  int base_ii = $row * blksz;
  int base_jj = $col * blksz;
  int base_kk;

  for (gg = 0; gg < aa.cols / blksz; ++gg) {
    base_kk = gg * blksz;

    // (10) These loops are shown unrolled to the right.
    for (ii = 0; ii < blksz; ++ii) {
      for (jj = 0; jj < blksz; ++jj) {
        for (kk = 0; kk < blksz; ++kk) {
          sum[ii, jj] += aa[base_ii + ii, base_kk + kk] *
            bb[base_kk + kk, base_jj + jj];
        }
      }
    }
  }

  for (ii = 0; ii < blksz; ++ii) {
    for (jj = 0; jj < blksz; ++jj) {
      cc[base_ii + ii, base_jj + jj] = sum[ii, jj];
    }
  }

  return cc;
}
```

```
\\ Unrolling results for blksize = 2

sum[0, 0] += aa[base_ii, base_kk]
           * bb[base_kk, base_jj];
sum[0, 0] += aa[base_ii, base_kk + 1]
           * bb[base_kk + 1, base_jj];
sum[0, 1] += aa[base_ii, base_kk]
           * bb[base_kk, base_jj + 1];
sum[0, 1] += aa[base_ii, base_kk + 1]
           * bb[base_kk + 1, base_jj + 1];
sum[1, 0] += aa[base_ii + 1, base_kk]
           * bb[base_kk, base_jj];
sum[1, 0] += aa[base_ii + 1, base_kk + 1]
           * bb[base_kk + 1, base_jj];
sum[1, 1] += aa[base_ii + 1, base_kk]
           * bb[base_kk, base_jj + 1];
sum[1, 1] += aa[base_ii + 1, base_kk + 1]
           * bb[base_kk + 1, base_jj + 1];
```

Listing 2: Blocked Matrix Multiplication in Bacon C

code, including more aggressive constant propagation and possibly static register-load scheduling enabled by the Bacon pre-optimizations.

The implementation of Bacon is available publicly under an open source license. The current version can be downloaded from the public git repository[1].

VI. Ease of Use

We believe that Bacon provides an improvement in ease of use compared to OpenCL by itself. A number of syntactic improvements are described in Section III. More significantly, the automatic generation of wrapper code performed by the Bacon compiler significantly reduces the number of lines of code that the developer is required to write.

As a concrete example of the reduction in lines of code required by Bacon, we consider naive matrix multiplication kernels like the one shown in Listing 1. Our OpenCL C version of this kernel is the same length: 17 lines of non-whitespace code. Unfortunately, even ignoring the application-specific code required to construct the input matrices, it takes 174 lines of non-whitespace C++ code to compile and call this 17 line OpenCL kernel. In contrast, the Bacon version only requires three lines to load, specialize, and call; all the bookkeeping is done either by the automatically generated wrapper code or by the Bacon runtime library.

[1] http://code.ferrus.net/compilers/bacon.git

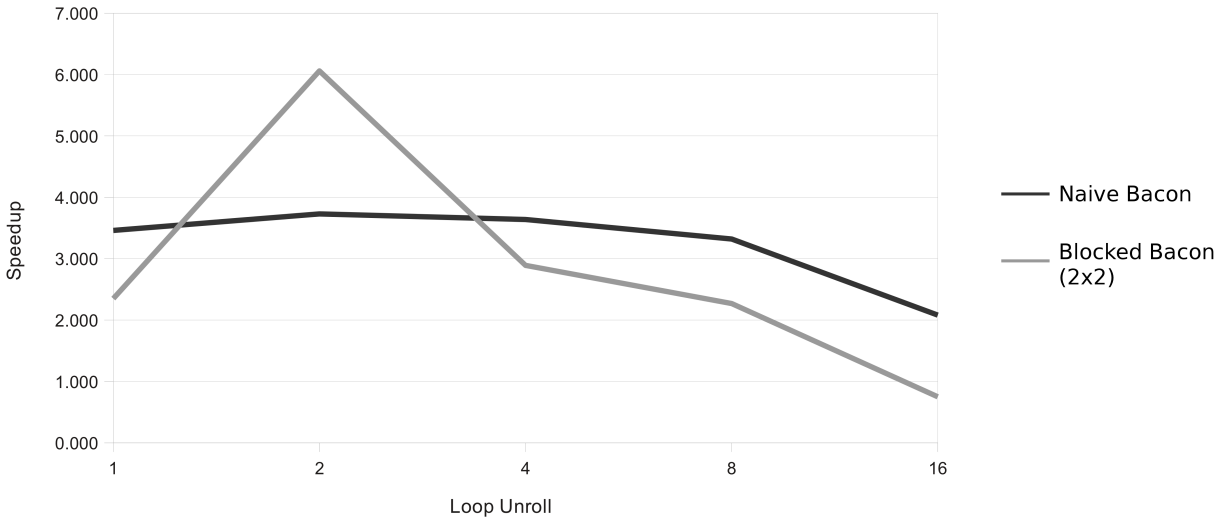


Fig. 1: Speedup on 4k matrix multiplication with unrolled loops over naive OpenCL implementation.

Test	Time (s)	Speedup
OpenCL - Naive	11.9	1.0
OpenCL - Hand Vectorized	2.54	4.7
Bacon - Naive (Best)	3.45	3.5
Bacon - Blocked (Best)	1.97	6.1

TABLE II: Summary of 4k Matrix Multiplication Performance

VII. Performance

In order to evaluate the performance of the Bacon system, we compare the run time of matrix multiplication kernels written in both Bacon C and OpenCL C. These results are summarized in Table II, which shows that Bacon is able to provide measurable performance improvements over similar programs written directly in OpenCL C.

Testing was performed on an AMD Radeon HD 5830 GPU. This is a mid-range GPU intended for high definition computer gaming. At the time of this writing, the card is a full hardware generation out of date, but it still has a theoretical parallel compute capacity of 1.7 teraflops, which is more than an order of magnitude greater than a high end CPU like the Intel Core i7 3930 at 154 gigaflops[2].

Four implementations of matrix multiplication were tested:

Bacon - Naive
: A textbook implementation of parallel matrix multiplication. Shown in Listing 1.

OpenCL - Naive
: An OpenCL implementation equivalent to the naive Bacon code.

Bacon - Blocked
: Shown in Listing 2. This generalizes a 2D unrolling of the computation into square blocks.

[2]FLOPS ratings from manufacturer specifications

OpenCL - Hand Vectorized
: Hand unrolled to compute 4x4 blocks at once. Explicitly uses OpenCL's native vector types. Based on a sample from the AMD OpenCL SDK.

The execution time of these four kernels was tested on randomly generated 4096x4096 matrices. Each test was performed five times and the average result was taken. The times were very consistent; most tests had a coefficient of variation under one percent. The speedups over the naive OpenCL implementation are shown in Figure 1.

A. Discussion

These measurements show that value specialization provides significant speedups over a non-specialized OpenCL kernel. This result can be explained by the fact that providing constant values and unrolled loops at compile time allows the OpenCL C compiler to do extensive constant propagation-based optimizations.

Unlike constant propagation, which provides a clear and consistent performance benefit, loop unrolling is more complicated. Unrolling loops too much drastically decreases performance, most likely due to the exhaustion of registers on the GPU device. Still, when properly tuned, loop unrolling provides significant speedups allowing the blocked Bacon kernel to beat the hand vectorized OpenCL kernel by nearly 30 percent.

Somewhat surprisingly, Bacon's loop unrolling is able to beat the vectorized OpenCL code without the explicit use of native vector types. From this we conclude that either vectorization is being done automatically by the AMD OpenCL compiler or the use of vector types doesn't have a significant performance benefit for this matrix multiplication algorithm.

VIII. Future Work

The Bacon system could benefit from a number of extensions to improve its performance, generality, and utility

to developers. One obvious area of improvement would be to automatically determine the best factor for loop unrolling rather than expecting it to be tuned manually. The unrolling analysis could also be extended to the implicit parallel "loops" created by the range of the computation; optimally, it should be possible to generate kernels like the blocked matrix multiplication shown from the naive matrix multiplication kernel. Another simple improvement would be to perform strip mining (Wolfe [10], section 9.8) and generate explicit vector code for loops, but this may be unnecessary if the OpenCL compilers are doing it themselves.

Just in time specialization as a method to improve performance on parallel computations seems very promising in general, but there are some limitations imposed by the use of OpenCL as a compiler target. An implementation of this technique targeting a parallel processor at a lower level would allow for more flexibility and much faster specialization times.

IX. Conclusion

We have shown that Bacon allows a high performance GPU compute kernel to be written in a naive style and executed with nearly the performance of a hand-vectorized OpenCL kernel due to the performance benefits of just in time specialization. Further, we have shown that by re-writing that kernel with a generalized block strategy and selecting appropriate loop unrolling settings, the performance of a hand-unrolled OpenCL kernel can be exceeded.

We are distributing this tool in the hope that it will be practically useful for developing kernels targeting OpenCL compatible GPU devices.

References

[1] "The OpenCL Specification 1.1." [Online]. Available: http://www.khronos.org/opencl/

[2] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé, "Tempo: Specializing systems applications and beyond c," *ACM Comput. Surv.*, vol. 30, September 1998.

[3] C. Chambers, "The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages," Stanford University, Tech. Rep., 1992.

[4] C. Bolz, M. Leuschel, and A. Rigo, "Towards just-in-time partial evaluation of prolog," in *Logic-Based Program Synthesis and Transformation*, ser. Lecture Notes in Computer Science, D. De Schreye, Ed. Springer Berlin / Heidelberg, 2010, vol. 6037, pp. 158–172.

[5] R. Webber, "clUtil - making OpenCL as easy to use as CUDA (Website)," http://forums.nvidia.com/index.php?showtopic=188713\&pid=1199058\&st=0, 2011, archived at http://www.webcitation.org/5z3zH1mtA on May 30, 2011.

[6] M. Hutter, "JOCL - Java bindings for OpenCL (Website)," http://www.jocl.org/, 2011.

[7] S. Ross-Ross, "Clyther opencl interface for python (website)," http://clyther.sourceforge.net/, 2011.

[8] O. Chafik, "Scalacl optimizing compiler plugin + gpu-backed collections (website)," http://code.google.com/p/scalacl/, 2011.

[9] F. Desarmenien, *Parse::Yapp*. [Online]. Available: http://search.cpan.org/~fdesar/

[10] M. Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

A Research of MapReduce with GPU Acceleration

Miao Xin[1], **Hao Li**[2], **Joan Lu**[2]
[1] School of Information Science and Engineering, Yunnan University, Kunming, Yunnan, China
[2] School of Software, Yunnan University, Kunming, Yunnan, China
[2] School of Computing and Engineering, University of Huddersfield, Huddersfield, West Yorkshire, U.K

Abstract - *MapReduce is an efficient distributed computing model on large data sets. The data processing is fully distributed on huge amount of nodes, and a MapReduce cluster is of highly scalable. However, single-node performance is gradually to be a bottleneck in compute-intensive jobs, which makes it difficult to extend the MapReduce model to wider application fields such as large-scale image processing and image mining. As an attempt, this paper presents an approach of GPU-accelerated MapReduce, which is implemented by Hadoop and OpenCL. Being a distinctive feature, it aims at general and inexpensive hardware platform, and it is seamlessly integrated with Apache Hadoop, the most widely used MapReduce framework. As a heterogeneous multi-machine and many-core architecture, it targets at both data- and compute-intensive applications. An almost 2 times performance improvement has been validated, without any special optimization.*

Keywords: MapReduce; GPU acceleration; Hadoop; OpenCL

1 Introduction

MapReduce is a distributed computing model oriented at huge datasets. Proposed by Google, it has been successfully applied to Google`s large-scale data processing [1]. For its excellent price/performance characteristic on low-cost clusters, nowadays, various MapReduce implementations are being utilized by more than 150 IT companies, making the data processing much faster, easier and more efficient [2].

MapReduce model simplify the complex parallel computing problem into two simple steps. First is called *Map*. The master node divides the input into smaller sub-problems, and distributes them to mapper nodes. Next step is *Reduce*. Reducer nodes collect answers to all sub-problems and combine them in some way to form the final output. All works across nodes are in highly parallel.

Among various MapReduce applications, compute-intensive tasks is an important class of applications, which is characterized by the fact that execution of the map function is significantly longer than the data accessing time, by an order of magnitude at least. In the case that the cluster scale is limited, some nodes have to process a large number of key/value data. This situation may be more serious during the Reduce stage of some programs, which is always the bottleneck of the whole system.

In order to handle this issue, conventional solutions have been presented:

- Scale-out [3, 4]: Adding more Mapper or Reducer nodes, artificially or automatically. Sometimes it works. However, the cluster scale`s expanding will pose some challenges, such as side-effects in scheduling and node faults, especially when the cluster scale is larger than 1000 nodes.
- Parameters optimization [5]: By adjusting the proportion of Mappers and Reducers, the file block size, data compression, etc., the cluster can get more perfect performance. Performance optimization is very useful in same cases; however, the ability of optimization is limited. This process is often time-consuming, and it needs skills. Moreover, one such optimization works on only one or one kind of jobs. That means it has to be re-turned when the program is changed.

Recent efforts therefore mainly focus on improving the availability, reliability and scalability for super large cluster, and cluster performance testing, etc. Researches are relatively less in the aspect of farther increasing the parallel computing capacity of single-nodes. However, the computing capacity within a node is a critical factor for performance improvement besides communication efficiency [6].

Analyzed the program of compute-intensive tasks, we can find some characteristics. It often contains many nesting loops in these programs, which often consume major running time. Nevertheless, sometimes the data dependency is quite low between loops, which determine that it is feasible to assign one loop to one thread, and execute these threads in parallel.

Heterogeneous computing architecture, a state-of-the-art tendency of high performance computing pattern [7, 8], has been emerged in recent years. Up to the end of 2011, three of the 5 most powerful supercomputers in the world take advantage of GPU acceleration. Comparing with multicore

CPUs, concurrent threads in GPUs are far more than CPUs. Hundreds of lightweight threads can be easily launched in concurrent [9]. And this number, with the development of chip technologies, will continually increase without the limitation of Moore`s law. GPUs are much good at parallel numerical calculation and steaming processing. GPU computation is of huge potential in compute-intensive tasks although it has not been completely released yet. Therefore, utilizing computing power of GPUs to enhance the single-node performance is our basic motivation.

Original MapReduce model targets at multi-machine. Hence, some implementation frameworks port on large cluster via distributed file system, e.g., the Google-MapReduce framework upon Google File System (GFS) and Apache Hadoop framework upon Hadoop File System (HDFS). In addition, some implementations extend it to multicore CPUs and shared-memory multiprocessors architecture, such as the Phoenix framework [10] proposed by Stanford University. Although these are all excellent implementation, they didn`t take huge potential of GPUs acceleration into consideration.

Related research didn`t bog down here. Some achievements have been proposed yet, such as:

- B.He et al. [10] presented an SMS-based MapReduce framework: Mars. It is implemented by NVIDIA CUDA and Phoenix [11].
- K.H.Tsoi and W. Luk [6] presented a MapReduce framework with mixed hardware accelerator: Alex, a heterogeneous cluster with FPGAs and GPUs.

Nevertheless, one limitation is that they all need dedicated hardware or bind with software platform, which makes it difficult to port on general and inexpensive platforms.

This paper presents our early work of an approach of MapReduce with GPU acceleration on general platform, which is implemented by Hadoop framework [12] with Open Computing Language [13]. We attempt to extend the parallelism of MapReduce model, from multi-machine alone to multi-machine with many-core, so as to enlarge the application domains of Hadoop framework, e.g. large-scale image processing. Its performance has been validated by our experiments.

The rest of this paper is organized as follows. Section 2 provides an over-view of MapReduce and GPGPU computation. In section 3, we take mathematics method to evaluate the ideal acceleration performance, and then the architecture and implementation are presented with the experiment. Our experimental results are presented in section 4. Finally, we conclude this paper and outline our future work.

2 Preliminaries

2.1 MapReduce model and Apache Hadoop

The original MapReduce model is proposed by Google for the purpose of supporting its critical services such as web search, log analysis, data mining, etc. in petabyte-level datasets. Apache Hadoop-MapReduce project is an open-source implementation. Hadoop is not only the most popular and widely used MapReduce framework by masses of enterprises, but also a distributed store and compute platform. It is designed as orienting at inexpensive commodity hardware, but it is of excellent fault-tolerance mechanism so as to successfully support super large clusters and petabyte-level data processing.

Being a programming model, MapReduce allows programmers to write functional-style code that is automatically parallelized and scheduled in a distributed system. Although its implementation is quite complex, a MapReduce program is rather easy to understand in logical. As is shown in Figure 1, a big input dataset is split into small chunks, and then stored in the distributed file system (e.g. HDFS). Each Map task can process a local input split. The intermediate outputs are transferred to Reduce nodes, then reduce tasks combine them in some way to form the final output.

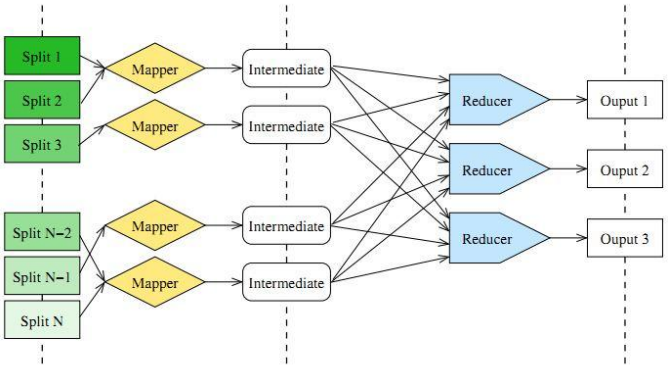


Figure 1. A Basic Process of a MapReduce task (Diagram courtesy of Apache Hadoop.)

Utilizing the distributed file system, data-locality optimization [14] maximizes the amount of data throughput, which makes the processing of data-intensive applications much faster. However, although the sub-tasks` running are in parallel, the default program running within each node is still in serial, that is inefficient comparing with multi-thread programs.

Therefore, improving the program parallelism of single-node is critical if we need farther performance improvement, especially when we are facing compute-intensive applications such as text processing, data compression, digital graphics processing, etc. In CPUs programing, multi-thread

programing is of difficulties and its performance improvement is rather limited because of CPUs` architecture.

Nevertheless, the situation is quite different in GPUs.

2.2 GPGPU and OpenCL

The Graphic Processing Unit (GPU) is a many-core processor that can concurrently execute hundreds of threads. Originally, it was a dedicated processor for graphics processing acceleration. But recently, being an extension of stream processor, the General Purpose Graphics Processing Unit (GPGPU) turns the massive floating-point computational power of a modern graphics accelerators` shader pipeline into general-purpose computing power [15].

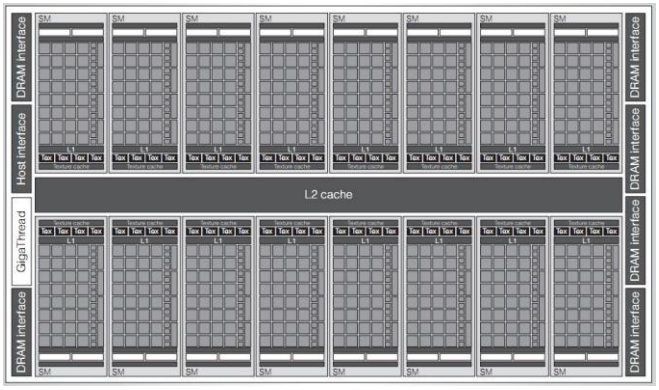


Figure 2. Fermi GPU architecture; 8 cores each streaming processors, and 16 processors each chip (Diagram courtesy of NVIDIA.)

For the ease of development, some GPGPU programing framework have been developed, such as NVIDIA CUDA, AMD CTM, and Brook+ developed by Stanford University, etc. They are all bind with appointed hardware or programing language.

Different from these, the Open Computing Language (OpenCL) aims at the general-purpose multicore computing. Initially sponsored by Apple Inc., OpenCL is the state-of-the-art open standard for cross-platform, parallel programming of modern processors found in PCs, servers and embedded devices. OpenCL gives any application access to the GPGPU for graphical or non-graphical computing.

An OpenCL program usually consists of two parts:

1) OpenCL Kernel: Kernel is the program that runs on multicore devices (e.g. GPUs). It is written by the OpenCL language, which is similar to C language.

2) Platform-control program: For the purpose of control the OpenCL runtime environment, a suit of API is necessary. It has several language bindings that can be chose by programmers, such as C, C++, Java, etc.

3 GPU acceleration and Algorithm

The substance of GPU acceleration is multi-threaded execution on multi-core hardware. There is usable scope that GPU acceleration is not applicable to all programs, although most MapReduce jobs are of natural fitness with GPU acceleration. We can use mathematics method to evaluate the acceleration performance and then validate it by experiments.

3.1 Acceleration theory

A MapReduce program is always both data- and compute-intensive. Commonly, there are many loop operations in it, especially in the *reduce* function. Owing to the large scale of input dataset and the complexity of data processing, the time consuming caused by loops is much longer than the other operations. In practice, supposed that if there is a low data dependency between each loops, codes inner the loop can be executed as an independent thread, and each thread processes different input data.

This is the Single Instruction, Multiple Data (SIMD) [16], which is the basis of GPU acceleration. The stream processors (SMs) in GPUs are far more than CPUs, so we can assign each small task on one thread, 100x decreasing the total executing time.

The Relative Speedup (Sp) [17] can well measure the acceleration performance. Relative speedup refers to how much faster a parallel improvement is.

In logical, a program can be divided into two parts. The first is all codes executed in sequential, which is denoted by S here; the second part is k m-dimensional loops that are denoted by L.

So, the running time of the whole program is $T = T_s + T_l$.

Suppose that the average executing time for each program statement is t, and there are n statements in each loop. Then the time consuming of the loop operation is:

$$T = T_s + k \times (n \times t)^m \qquad (1)$$

Using GPU acceleration, we suppose the quantity of concurrent threads is N, then the executing time is:

$$T_a = T_s + \frac{k \times (n \times t)^m}{N^m} \qquad (2)$$

When Tl is long enough, Ts is negligible. Thus, the related Sp for an m-dimension loop is:

$$Sp(m) = \frac{T}{T_a} = N^m \qquad (3)$$

We can see that the relationship between speedup and the quantity of nesting-level of loops is positive correlation.

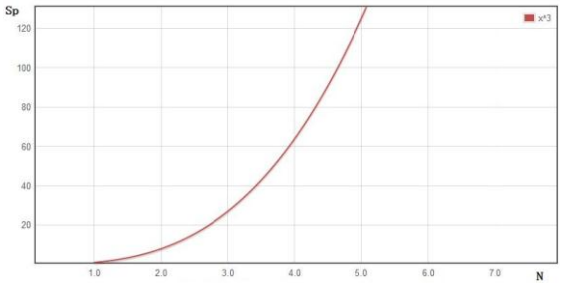


Figure 3. The relationship between Sp and N; (m = 3)

Parallel acceleration can be used to reduce the running time of programs with many parallel-able loops. Hence the GPU acceleration should be applied to heavy and repeated tasks. However, logical transaction functions of GPU are limited, i.e. GPUs can not substitute CPUs. Within our heterogeneous MapReduce node, CPU is in charge of program control, task distributing and I/O accessing, when GPU is responsible for intensive computing. CPUs and GPUs cooperate to accomplish a data- and compute-intensive application. We call it a heterogeneous computation cluster.

This method would help us evaluate the ideal performance improvement, which can be used to make initial decision about whether we would take advantage of GPU acceleration. However, the program should satisfy prerequisites of utilizing parallel acceleration.

3.2 Prerequisites of parallel acceleration

Analyzed from characteristics of SIMD and our experiments, prerequisites of utilizing GPU acceleration can be concluded as follows:

- The program contains many and/or high-dimensional loops: The amount of cycle number of loops should be far more than the number of CPU cores and close or greater than the number of concurrent threads in GPU. If the cycle number is too small, the loss outweighs the gain because the parallel environment context also needs time to initialize. But too big cycle number is not recommended either, e.g., the disparity between concurrent threads number and cycle number is larger than one order of magnitude. For this case, it should be considered about multi-GPUs or even larger cluster.
- There is no complex condition branches inner loop code: Flow-control-heavy tasks are not fit for GPUs. The performance loss caused by frequently cut-off between GPUs and CPUs can not be ignored either.
- The data dependency should be low enough: Ideal condition is that there is no synchronization or data dependency across iterations. In this case, we can gain fully data parallel. If this constrain can not be perfectly satisfied, there must be a tradeoff between the program complexity and performance improvement.
- Task type: Numerical calculation, text or graph processing are recommended.

First three are necessary conditions, when the last is an additional condition. If all can be well satisfied, then the GPU can play its most powerful effect.

3.3 The Algorithm of Performance Test

We have performed 3 kinds of experiments, including Matrix Multiply, String Match and KMeans [10], which are compute-intensive or data-intensive. Here we describe a typical test case which is both data-intensive and compute-intensive.

First, the algorithm description is given here:

LineFilter*: The input data is a very big text file (1GB, 10GB… 1TB), each line of which is consisted by 512 random digit from 0 to 9. For each group of five digits, it needs to calculate the medium value, and then substitute it to the first digit of the group. Keep doing this until the end of this file. Then statistics that how many the two adjacent numbers` difference is greater than 5. The results should be stored in an output file.*

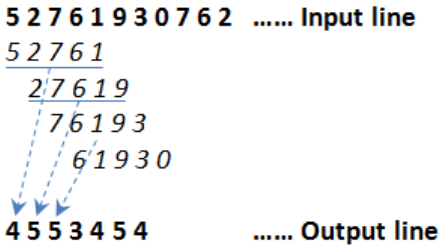


Figure 4. A Sample of DataTestMap

And here is the pseudo code:

Algorithm LineFilterMap:

```
LineFilterMap(key lineOffset, value line)
   line = data_formate(line)
    do until  the end of this line
    {
       for each line[j]…line[j+4] belong
to this line
```

```
      result = (line[j] + line[j+1] +
               line[j+2] + line[j+3] +
               line[j+4]) / 5
      line[j] = result
   }
   write(key, line)
```

Algorithm LineFilterReduce:

```
lineFilterReduce (key lineOffset, value
line)
   do untile  the end of this line
   {
      for each line[j]
      if( abs(line[j]-line(j+1))>=5)
num++
   }
   write(lineOffset, num)
```

This test is analog to the smoothing operation in digital image processing, which is a typical data- and compute-intensive task.

In the *lineFilterMap* function, there is a 1-dimensional loop, which will be executed 512 times in linear on one compute-core. However, it can be founded that the sub-dataset is not intersectional between each loop. That is data independence. This fine characteristic permits that it can be executed in parallel.

3.4 Implementation details

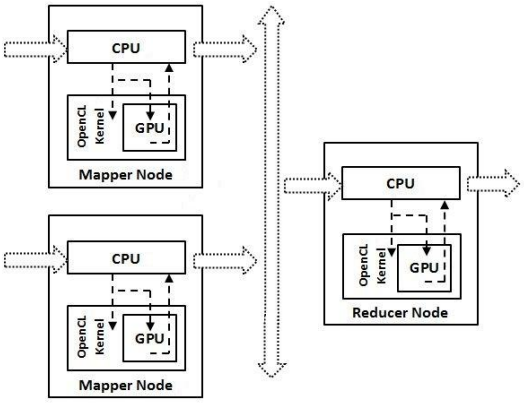


Figure 5. The Data Flow of the Architecture

When performing the GPU-accelerated program design, one of the most important questions is deciding which part of the program should be assigned to GPUs. Hence, we provide our recommended program design principal according to our experiments.

Responsibilities of CPUs are concluded as follows:

- Program control: Main frame of a MapReduce program.
- Tasks distributing: Launching the compute-intensive part to the GPU, including initializing OpenCL context, kernels establishing, data interchange between host memory and device memory, and OpenCL objects collection.
- Data accessing: I/O operations, including data accessing from local disk or distributed file system.

In our implementation, we suggest that a map or reduce task should be executed by several CPU threads, and each single CPU thread will spark lots of GPU threads. The programmable units of the multi-core CPU follow a Multiple Instruction, Multiple Data (MIMD) programming model. These few heavyweight but multifunctional CPU threads can execute much more complex tasks.

Comparatively, the functions of Kernels are much simpler. GPU focuses on intensive computing and efficiency. Considering the performance, there should be not any I/O operations in Kernels. For the same reason, the frequency of data transfer between host memory and device memory should be designed as low as possible [18].

The real process of *LineFilter* program is demonstrated as follows:

When the context of Hadoop-MapReduce runtime is initialized in the whole cluster, each data node (or *TaskTracker*) has got its task and data. In each Map program, there are two CPU sub-threads in Map function: First is to initialize the OpenCL platform context; another is to format input data into the form that can be processed by GPUs. When these two sub-threads have been done, the OpenCL kernel program begins to run on GPU. All these GPU computation results will be pushed into a high-bandwidth device buffer cache in parallel. Then, the results will be written to local disk.

Part of the Map Kernel:

```
int iGID = get_global_id(0);

if (iGID >= numElements)  {

      return;

} else {

   result[iGID] = ( a[iGID] + b[iGID] +
         c[iGID] + d[iGID] + e[iGID] )/ 5;

}
```

Next step is processing the massive intermediate outputs. Facing more than 2000000000 lines data, it is really a big job. However, with 3 Reduce nodes and 1000 threads per node running in our experiment environment, this scale is reduced to be less than 67000. The structure of Reduce function (lineFilterReduce) is similar to the Map function. It also

include OpenCL acceleration, however, if the Reduce node is coincidentally also the Map node, the time of OpenCL context initializing has been designed to be saved, since the kernel function of Reduce is integrated into the kernel file, and that is also initialized when the Map function is started.

4 Experiments

For getting an intuitionistic comparing between the native MapReduce and our improving, experiments have performed.

4.1 Experiment environment

The experiment environment is deployed on a 7 nodes cluster of DELL commodity PCs. The cluster statistics are:

- 7 nodes and 3 nodes per a rack
- 1 dual core Xeons @ 2.0Ghz per a node
- 2G RAM per a node
- 1 ATI Radeon HD 5450 GPU per a node
- 1 SATA disk per a node
- 1 Gigabit Ethernet on each node
- 8 Gigabit Ethernet uplinks from each rack to the core switch
- Ubuntu11.10 Enterprise Linux Server
- Oracle Java JDK 1.7.0_01-b08
- Hadoop 0.20.203
- OpenCL 1.1

Performance and compatibility are the main considerations. For the purpose of reaching a native performance, the Hadoop framework in this test is deployed on bare nodes without virtualization or hypervisor. The distributed file system is also the native HDFS.

Since the Java language is the best practice in Hadoop programing, for achieving a better seamless-integration, we select an Java language binding (JOCL) of OpenCL to integrate these two frameworks together. JOCL use Java Native Interface (JNI) to call the kernel program that drives the GPUs. Mixed programming guarantees their well-integrated meanwhile the performance loss will be the least.

4.2 Results

Under the different orders of magnitude, we have completed 5 groups of experiments. The MapReduce program with GPU acceleration is denoted by *MRCL*. Each group includes the performance comparison: the native MapReduce program, denoted by *MR*. Each group has been run for 5 times. The average value is recorded. Final result of experiment 1 is listed as follows:

TABLE I. RESULTS OF EXPERIMENT 1 ON DIFFERENT INPUT SCALE

Input Scale	Performance		
	MR Elapsed time	***MRCL Elapsed time***	***Sp***
1GB	3m:26s:159ms	2m:8s:373ms	1.606
10 GB	16m:51s:073ms	9m:12s:507ms	1.797
100 GB	53m:03s:112ms	28m:17s:331ms	1.875
1 TB	223m:39s:581ms	115m:59s:026ms	1.942

As is shown in this table, the speedup is all greater than 1.6 and it is gradually close to 2 with the input scale`s increasing. However, the program in this test only contains a 1-dimentional loop. If there are more high-dimensional loops, we can expect an even larger speedup.

Furthermore, in all test cases of experiment 1, the map function gets only one line of data each time. It is the simplest manner, but it is not the best practice. Because it will lead to "GPU hunger" and the overhead caused by frequent data transmission by PCI-E is considerable. A better choice is to submit more data per time, e.g. 5 lines each batch. But this number is not the more the better. Hence, we perform the second experiment to quest the optimized number of batch submission to GPU.

We override the RecordReader class of Hadoop framework. Then we perform the experiment 2 under 1 GB input scale. In figure 6, we can find that the best case appears when 3 lines (3×512 bytes) of data are submitted to GPU per one batch.

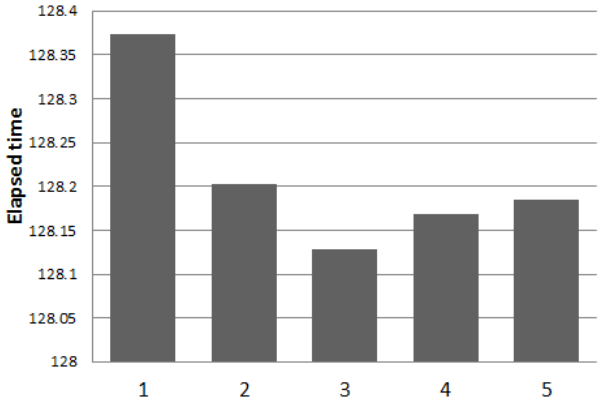


Figure 6. The comparison of elspsed time in different batch

We therefore perform the experiment 1 again (denoted by experiment 3) using the new *RecordReader* class. Figure 7 shows the peak speedup of experiment 3 is 1.959.

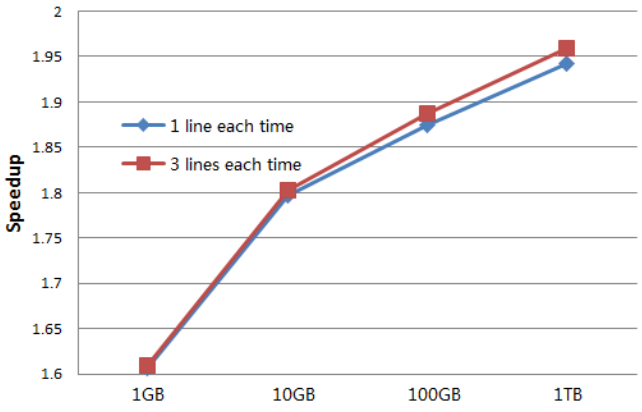


Figure 7. Performance Speedup of the Experiment 3 over the Experiment 1 counterpart.

It has to be noted that, to obtain an original contrast, there is not any special optimization in MRCL program. The structure of MRCL program is almost the same with the MR program except the part of GPU acceleration. That means there is still considerable improvement space.

5 Conclusions

Big data processing requires both high-capacity distributed storage and higher cluster computing performance. In this paper, we proposed an approach to take advantage of GPUs for reaching a higher single-node performance. An outstanding feature is that it is platform independence and can be seamless-integrated with existing framework. Utilizing Hadoop and OpenCL, a low-cost computing cluster can achieves more powerful computing capacity. With an almost 2 times performance improvement, we can expect it will be applied in both data- and compute-intensive applications.

Current work is still the technology demonstration, which has validated the feasibility of this technology roadmap. Future work will be focused on framework integration for the ease of development and farther performance improvement.

ACKNOWLEDGEMENT

This work is supported by: The National Nature Science Foundation of China 2010 No.61063044; The Key Discipline Foundation of School of Software of Yunnan University and the open Foundation of Key laboratory in Software Engineering of Yunnan Province under Grant No.210KS05; The Science and Technology Department Nature Science Foundation of Yunnan Province, China 2011 No.2011FZ014; The Education Department Key Projects of Nature Science Foundation of Yunnan Province, China 2011 No.2011Z030.

The authors thank the anonymous reviewers for their insightful suggestions. We further wish to thank EMC Labs China, OpenCL and Apache Hadoop open-source community for their illustrious works.

6 References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] Clifford Lynch, "Big data: How do your data grow" Nature 455: 28–29, 2008.

[3] R. Kumar, V. Zyuban, and D. M. Tullsen. "Interconnections in multicore architectures: Understanding mechanisms, overheads and scaling." In Proceedings of International Symposium on Computer Architecture, 2005.

[4] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. "Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors." In Computer Architecture Letters, Vol 4, July 2005.

[5] Performance of Map Reduce job. http://blog.dynatrace.com/2012/01/25/about-the-performance-of-map-reduce-jobs/. 2011

[6] Kuen Hung Tsoi and Wayne Luk. "Axel: a heterogeneous cluster with FPGAs and GPUs." In FPGA '10: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays, pages 115{124, New York, NY, USA, 2010. ACM.

[7] T. Endo and S. Matsuoka. Massive supercomputing coping with heterogeneity of modern accelerators. In IEEE International Symposium on Parallel and Distributed Processing, IPDPS'08, pages 1-10, 2008.

[8] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "MapCG: Writing parallel program portable between CPU and GPU," in Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, Sep. 2010, pp. 217–226.

[9] J. Nickolls and W. J. Dally. The gpu computing era. IEEE Micro, 30(2), 2010.

[10] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, Oct. 2008, pp. 260–269.

[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Apache Hadoop. http://hadoop.apache.org/, 2011.

[13] OpenCL. http://www.khronos.org/opencl/, 2011

[14] MCKINLEY, K.S., CARR, S., AND TSENG, C.-W. 1996. "Improving data locality with loop transformations." ACM Trans. Program. Lang. Syst. 18, 4 (July), 424–453

[15]] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," Proceedings of the IEEE, vol. 96, pp. 879–899, 2008

[16] K. Knobe, J. D. Lukas, and G. L. Steele. "Data optimization: Allocation of arrays to reduce communication on SIMD machines." Journal of Parallel and Distributed Computing, 8:102–118, 1990.

[17] M. Flynn, R. Dimond, O. Mencer, and O. Pell. Finding speedup in parallel processors. In Proc. Int. Symp. on Parallel and Distributed Computing ISPDC'08.

A top-down algorithm for clustering in large-scale distributed networks

A. Bui, S. Clavière and D. Sohier
PRiSM Laboratory, CNRS, Université de Versailles St-Quentin en Yvelines, Versailles, France

Abstract— *In this article, we propose a distributed top-down clustering algorithm for large-scale systems subject to topological changes. The objective of this clustering is to divide large systems into smaller subsystems on which distributed algorithms can be executed efficiently. The main parameter of distributed algorithms complexity is the size of the system, which leads us to focus on bounded-size clusters. Top-down approach allows us to build a cluster naming from which we derive an efficient inter-cluster routing scheme. Clusters are built by a token moving from node to node according to a random walk scheme. When the cluster reaches a maximal size defined by a parameter K, it is divided (when possible) and tokens are created in both of the new clusters. The new clusters are then built and divided in the same fashion. A spanning tree is built during the construction. These spanning trees, together with the edges along which clusters have been divided, constitute a spanning tree of the system, which allows routing thanks to a consistent routing scheme. In a dynamic system, the clustering may be divided into two disjoint structures. This is also the case when multiple initiators begin the clustering process. When two structures meet, one is "adopted" by the other. This adoption mechanism allows the clustering to adopt transparently to topological changes*

Keywords: Distributed algorithms, hierarchical clustering, bounded-size clusters, dynamic graphs, random walks

1. Introduction

Large-scale distributed systems raise new challenges to distributed computing. In this paper, we focus on dividing a large-scale distributed system into smaller subsystems that can be managed separately and coordinated. We are willing to compute size-oriented clusterings: given a target size K which is a parameter of the algorithm, clusters should all have size roughly K. More precisely, we would like that all clusters are connected and have a size between K and $2K$ (allowing for a process considering only one cluster to detect when it is too large and to fix it).

However, such a clustering cannot be achieved on some topologies. Consider for instance a star graph on $M > 2K$ nodes. The central node is in a cluster $\mathcal{C}$. Then, either all nodes are in $\mathcal{C}$, which breaks the requirement, or some clusters are of size 1, which is also contradictory to the specification. This is why we introduce the notion of divisible cluster: a cluster is divisible if can be divided into two connected clusters of size greater than K. The clustering we are computing is such that all clusters are of size greater than K, and none is divisible. It can be seen as a distributed approximation of a clustering with all clusters of size K.

Since the algorithm is based on the size of clusters, each cluster must be *"aware"* of its size. Thus, nodes should be recruited one by one, and to model this, we use a recruitment token. The token must follow a traversal scheme to ensure that all nodes can be recruited. Since no assumption is made on the topology of the system and the topology is unknown and possibly dynamic, we use a random walk traversal scheme [14]. Properties of random walks allow to design a traversal scheme using only local information and to ensure local load balancing; such a scheme is not designed for one particular topology and needs no adaptation to fit other ones [6], [9]. In the course of its walk, the token builds a spanning tree of the cluster which it uses to divide the cluster when required. This tree is also the basis for routing messages in the cluster, and between clusters.

Each cluster is then represented by its recruitment token, that stores all information relevant to it (in particular, this spanning tree and inter-cluster routing information).

Thus, the walk recruits unclustered nodes and maintains an adaptive spanning tree of them, until it reaches size $2K$. At this point, the token holder checks whether the tree is divisible, and if it is the case, divides the cluster. Since the spanning tree is adaptive, if the cluster is divisible, the tree is eventually divisible too. The division process is carried out by the propagation of a wave on the spanning tree. Each node changes its cluster id accordingly, and the roots of the two subtrees create new tokens, while the former one is discarded. The edge along which the division was performed is recalled by nodes of both clusters, and will be used for inter-cluster routing. We call these edges inter-cluster routing edges (ICRE in the following).

This algorithm is top-down. This choice allows us to design a naming scheme for clusters to efficiently route messages between clusters. Each cluster has a name in $(0|1)^*$. During a division, the resulting clusters take the name of the divided cluster, and add a 0 or a 1 at the end. Thus, two clusters the names of which differ only by their last bit are siblings, and know a link to each other. They may be further divided, but one of the descendants of each of them will keep track of this link, thus allowing inter-cluster routing. Other descendants will recall which sub-cluster holds this link.

Clusters are named in the course of the divisions: initially, an initiator node starts building cluster 1. When a cluster with id x is divided, it leads to two new clusters, one named $x0$, and the other $x1$. x is then used as a name for the union of clusters $x1$ and $x0$; A cluster x is called *elementary* if $x0 = \emptyset$ and $x1 = \emptyset$. A non elementary cluster is called a *virtual cluster*. All clusters belonging to the same (virtual) cluster 1 form a *structure*. In a structure, the level of a cluster is the length of its id.

This naming scheme, together with the inter-cluster routing edges computed during the divisions, provide the basis for inter-cluster routing. The set of all cluster tree edges and inter-cluster routing edges form a spanning tree of the structure.

In the event of a topological change, if this change does not affect a cluster tree or an inter-cluster routing edge, the algorithm goes on unaffected. If the change discards a cluster tree edge or a inter-cluster routing edge (for instance, if a clustered node disappears), we have no structure spanning tree any longer, and we cannot guarantee that the structure is connected. Then, the structure is divided into two: a wave is broadcast along its tree allowing a consistent renaming of clusters.

In such a situation, or when there are several initiators, several structure grow in parallel. To ensure communication between their clusters, they have to be merged. This is realized by making one of them *"adopt"* the other. To avoid mutual adoption, a total order between structures is introduced (based on the ids of the nodes in their first division edge). When a walk in the weaker structure reaches a node in the stronger one, it broadcasts an adoption wave on its structure to make it a substructure of the stronger structure. More precisely, if the walk hits a node in cluster x, cluster x is renamed $x0$, and all clusters in the adopted structure add x as a prefix to their cluster name, so that the adopted structure becomes the $x1$ substructure of the structure.

All communications that occur inside a structure during an absorption process are carried out transparently.

1.1 Routing in a binary hierarchy of named clusters

For the sake of simplicity, we do not distinguish between a cluster and its id.

We consider a binary hierarchy of clusters such that:

- all nodes have a binary cluster id (clusters $x0$ and $x1$ are called siblings; their reunion, with id x, is called their parent);
- nodes with a given cluster id prefix form a connected subgraph;
- each elementary cluster (ie cluster with no sub-cluster) has a spanning tree (local tree in the following);
- given a non-elementary cluster x, there exist a node i in $x0$ and a node j in $x1$ such that i and j are neighbors, and all nodes in $x0$ are aware of $(i(x0y), j)$ with $x0y$ a cluster to which i belongs, and all nodes in $x1$ are aware of $(j(x1z), i)$ with $x1z$ a cluster to which j belongs; we call (i, j) and inter-cluster routing edge.

The algorithm proposed in the next section, once all nodes are clustered in the same structure, provides such a binary hierarchy of clusters. For a more detailed explanation of the routing scheme below, see [7].

The set of edges of the local trees and of inter-cluster routing edges forms a spanning tree of the system, we call the global tree. It contains at most $n-1$ edges, since the local tree of a cluster of size k has $k-1$ edges, and each cluster shares one ICRE with its sibling; affecting the ICRE to the sibling that has not been affected the upper level ICRE, each elementary cluster has k edges except for one (the highest level ICRE is affected to only one of the highest level siblings). Hence there are $n-1$ edges in this subgraph.

We show in the following that routing can be achieved through the global tree, which proves that it is connected, and thus a tree.

Consider two different clusters x and y. Let z their longest common prefix (at worst, $z = 1$). Then, $x = zt$ and $y = zu$. By definition of z, u and t begin with different bits. Assume for instance that $t = 0t'$ and $u = 1u'$. Then, to route a message from x to y, we will first route it to $z1$. All this routing, except for its last step, will take place in $z0$. Once the message is in $z1$, either it is in y and the process is finished; or it is not, and then the same process will be used, but inside $z1$. This guarantees the progression of the process, and that the message is eventually delivered to the intended cluster.

To route the message to $z1$, the node looks at its ICREs. Consider $(i(z0v), j)$ the ICRE to $z1$. If $z0v = x$, then i is in x, and the local tree allows to compute a path to it. If $z0v \neq x$, then the message has to be routed to $z0v$. The longest common prefix between x and $z0v$ is longer than z, which allows to run this algorithm recursively.

No edge is used twice in this routing process; thus this routing process is a compact version of the routing tables that can be deduced from the global tree.

1.2 Related work

The algorithms of [4], [8], [12], [13] produce clusters with radius one. Each cluster has a node called a *clusterhead*, and all other nodes in that cluster are neighbors of the clusterhead. Clusters can be built using a hierarchical method: the clustering algorithm is iterated on the overlay network obtained by considering clusters as nodes, until a single cluster is obtained [11], [15], [17]. These solutions are bottom-up processes while we propose a top-down approach [16]. The algorithms in [5], [9] are based on random walks. In [9], the clusters are built around bounded-size connected dominating sets. In [5], the algorithm recursively breaks the network into two clusters as long as every cluster satisfies a lower bound.

The solution given in [3] builds k-hop clusters (clusters of radius k) and in [10], a self-stabilizing – a self-stabilizing system is a system that eventually recovers a normal behaviour after a transient fault – a $O(n)$-time algorithm is given for computing a minimal k-dominating set; this set can then be used as the set of clusterheads for a k-clustering.

1.3 Outline

Section 2 presents the algorithm. First, we present the mechanisms involved in the management of static networks with a unique initiator. Then, we present the reactions of the algorithm to topological changes. Last, we present the way to deal with multiple initiators building multiple structures, and how the algorithm merges them.

Section 3 gives a sketch of the proof of this algorithm, following the same progression.

In section 4 are provided some results on the complexity of this algorithm.

We conclude this paper by giving some conclusions and perspectives.

2. Algorithm

We consider a distributed system as an undirected connected graph $G = (V, E)$, where V is a set of nodes with $|V| = n$ and E is the set of bidirectional communication links with $|E| = m$. A communication link (i, j) exists if and only if i and j are neighbors. Every node i maintains a set of its neighbors ids denoted by N_i.

P_i is the id of the cluster the node i belongs to. Initially each node i sets $P_i = 0$ to indicate that it has not yet joined a cluster. These node are then qualified of *unclustered* node.

$Parent$ is an array describing the spanning tree of a cluster: $Parent_i[j]$ is the parent of j as known locally by node i; note that $Parent_i[i]$ is the current parent of node i, since a node changes its parent only when receiving the token, and then updates its $Parent$ array. Initially, each node i sets $Parent_i[j] = \perp$ for all j in V.

$ICRE_i$ is an array on node i containing all inter-cluster routing edges relevant to the cluster: consider $(a(x), b)$ as the k-th entry of $ICRE_i$. x is a prefix of a's cluster id. The route to the cluster with id consisting in the first $k-1$ bits of the name of the current cluster, and the k-th bit different goes through edge (a, b); to reach this edge, one must first route the message to x.

In the following, $ICRE[0]$ is used as a structure identifier to break symmetries (since it is known by all nodes in the structure). Before the first division, since this $ICRE$ is not yet defined, we set $ICRE[0]$ to the id of the initiator of the structure. The ordering is the following: a structure that not yet been divided is always smaller than one that has been; for structure that have been divided, $(i, j) < (i', j')$ iff $\min\{i, j\} < \min\{i', j'\}$.

Each time a division occurs, an ICRE is created, and the set of all ICRE creates a spanning tree of the clusters. The union of ICREs and cluster tree edges is thus a spanning tree of the whole structure, in which messages are routed.

All algorithms below are described on node i. The emitter of the message is triggering the algorithm is e.

2.1 Initialization

Nodes wake up either spontaneously, or upon reception of a message. If they wake up spontaneously, they are initiators and execute the following code.

Algorithm 1 On waking up (initiator)

1: $Parent_i[i] \leftarrow i$
2: $ICRE_i \leftarrow \{i\}$
3: $P_i \leftarrow 1$
4: Send $Token(P_i, Parent_i, ICRE_i)$ to $k \in N_i$ chosen at random
5: $Parent_i[i] \leftarrow k$

An initiator creates its cluster, with id 1, and sends a token message to one of its neighbors according to the random walk scheme. Then, it updates its local copy of the spanning tree $Parent$, taking into account that the node they sent the token to will be its parent as soon as it receives the token (see algorithm 2).

2.2 On reception of Token messages inside a structure

a) Algorithm 2: On reception of a Token message (unclustered node, or node in the same cluster): When an unclustered node receives a token, or a node receives a token from its own cluster, it joins the cluster, updates the spanning tree, checks its consistency with local topology (see 2.4), checks if the cluster should be divided with function $isDivisible$, and, in this case, initiates the division process, and sends the token to a random neighbor.

Function $isDivisible(Parent)$ returns either a node id corresponding to the root of one the subtree of tree $Parent$ which is divisible or -1 if tree $Parent$ is not divisible into two subtrees of size $\geq K$. The division process is detailed in section 2.3, in particular message $Child0$ and function $Sub0$ are defined.

Algorithm 2 On reception of *Token* $[P_t, Parent_t, ICRE_t]$ message with $P_i = 0$ or $i \in Parent_t$

1: **if** $P_i = 0$ **then**
2: {Join the cluster.}
3: $P_i \leftarrow P_t$
4: $ICRE_i \leftarrow ICRE_t$
5: **end if**
6: {Update cluster tree.}
7: $Parent_i \leftarrow Parent_t$
8: **if** $Parent_i[e] \neq \perp$ **then**
9: {$Parent_i[e] = \perp$ means that the token has been bounced by e which then does not belong to the cluster, see algorithm 3}
10: $Parent_i[e] \leftarrow i$
11: **end if**
12: $Parent_i[i] \leftarrow i$
13: {The next instruction is a call to Algorithm 7 checking if a topological change has outdated the spanning tree}
14: $LocalCheck()$
15: {Check whether the cluster is divisble, and initiates the division if necessary}
16: $d \leftarrow isDivisible(Parent_i)$
17: **if** $d \neq -1$ **then**
18: $P_i \leftarrow P_i.0$
19: **for all** $k \neq i$ such that $Parent_i[k] = i$ **do**
20: Send $Child0[Parent_i, d]$ to k
21: **end for**
22: $(Parent_i, ICRE_i) \leftarrow Sub0(Parent_i, ICRE_i, d, P_i)$
23: **end if**
24: {The walk resumes, and the node stores the next node it will visit}
25: Send $Token[P_i, Parent_i, ICRE_i]$ to k chosen at random in N_i.
26: $Parent_i[i] \leftarrow k$

b) Algorithm 3: On reception of a Token from a different cluster in the same structure: When receiving a token from another cluster in the same structure (two nodes i and j are in the same structure if $ICRE_i[0] = ICRE_j[0]$), the node ignores it and sends it back if the cluster is already of size greater than K. If the cluster is smaller than K (following a topological change), it should be deleted. The treatment of the message $Endcluster$ is given by Algorithm 8.

Algorithm 3 On reception of *Token* $[P_t, Parent_t, ICRE_t]$ message with $i \notin Parent_t$ and $ICRE_i[0] = ICRE_t[0]$

```
if |Parent_t| < K then
    Send EndCluster[e] to e
else
    Send Token[P_t, Parent_t, ICRE_t] to e
end if
```

2.3 Division

The division of a cluster along edge (k, j) is carried out through a wave on the tree: all nodes above j receive $Child0$ messages and switch to cluster $P.0$; all nodes in the subtree rooted in j receive $Child1$ messages and switch to cluster $P.1$. j receives a $Child0$ message and propagates $Child1$ messages to its descendants; then, it creates its new cluster token.

During a division, the list of ICRE has to be updated. Consider a cluster with id x, with an ICRE (a, b). Then, a is only in one of the sub-clusters, for instance $x0$. Then, $x0$ keeps the ICRE (a, b), and $x1$ updates its ICRE list by listing $(a(x0), b)$, meaning that to reach the ICRE (a, b) from $x1$, one must first send the message to $x0$.

Functions $Sub0$ and $Sub1$ compute the subtrees of the tree given as an argument obtained by removing the edge between node j (third argument) and its father. $Sub0$ computes the top subtree and $Sub1$ the bottom. The ICRE list is updated as explained above.

Top subtree

Algorithm 4 On reception of $Child0[Parent_t, j]$ message

```
if i ≠ j then
    P_i ← P_i.0
    for all k such that Parent_t[k] = i do
        Send Child0[Parent_t, j] to k
    end for
    (ICRE_i) ← Sub0(Parent_t, ICRE_i, j, P_i)
else
    P_i ← P_i.1
    for all k such that Parent_t[k] = i do
        Send Child1[Parent_t, j] to k
    end for
    (Parent_i, ICRE_i) ← Sub1(Parent_t, ICRE_i, j, P_i)
    Send Token[P_i, Parent_i, ICRE_i] to k ∈ N_i chosen at random
end if
```

Bottom subtree

Algorithm 5 On reception of $Child1[Parent_t, j]$ message

```
P_i ← P_i.1
(ICRE_i) ← Sub1(Parent_t, ICRE_i, j, P_i)
for all k such that Parent_t[k] = i do
    Send Child1[Parent_t, j] to k
end for
```

2.4 Topological changes

c) Algorithm 6: Topological change detection: The parent of a node i in the tree is $Parent_i[i]$. When a node detects that its parent is no longer a topological neighbor, it initiates the destruction of the cluster subtree rooted in it, since the cluster may not be connected anymore. The token is necessarily on the other side of the tree.

Algorithm 6 On failure of the link $(i, Parent_i[i])$

```
DestroyCluster()
```

The $LocalCheck$ procedure (*cf.* algorithm 7) checks the consistency of the tree with local topological information. If a node listed as a child of the current node in the tree is not one of its neighbors, the subtree rooted in the node that is no longer a neighbor is deleted from the tree. This may lead to the loss of some ICRE. In this case, $EndGate$ messages are propagated to make the structure consistent. If the cluster is smaller than K, either it grows and reaches size K, or it is eventually deleted (Algorithm 3).

Algorithm 7 $LocalCheck()$

```
for all (j ≠ i such that (Parent_i[j] = i) ∧ (j ∉ N_i)) do
    Parent_i ← Sub0(Parent_i, ICRE_i, j, P_i)
end for
for all (j such that (((l(P_i), j) ∈ ICRE_i)) ∧ l ∉ Parent_i) do
    {(l(P_i), j) is listed as an inter-cluster routing edge with l in the cluster, but l is not in the cluster}
    {Update local structure.}
    for all k ≠ i such that Parent_i[k] = i ∨ (i(P_i), k) ∈ ICRE_i do
        Send EndGate[(l, j), i] to k
    end for
    (P_i, ICRE_i) ← Prune((l(P_i), j), i)
end for
```

d) Algorithm 8: On receiving an *EndCluster*$[Node]$ message: $EndCluster$ messages are used to propagate the deletion of a cluster. If the message comes from its father, a nodes leaves its cluster. If the message comes through an ICRE, then the node updates its ICREs.

Algorithm 8 On reception of *EndCluster*$[Node]$ message

```
if Parent_i[i] = e then
    DestroyCluster()
else if ∃j/ICRE_i[j] = (i, Node) then
    UpdateGate(j, i)
end if
```

To propagate the destruction of its cluster (that can be initiated only by the token holder, ie the tree root), a node sends $EndCluster$ messages to its children in the tree and to nodes to which it is linked by an ICRE (so that they can update their ICRE list). Then it leaves the cluster and falls asleep. It will wake up when it receives a message, or spontaneously.

Algorithm 9 DestroyCluster()

1: **for all** k such that $Parent_i[k] = i \vee (i(P_i), k) \in ICRE_i$ **do**
2: {Propagate the changes in cluster.}
3: Send *EndCluster*$[i]$ to k
4: **end for**
5: $P_i \leftarrow 0$
6: Fall asleep

$UpdateGate$ (*cf.* algorithm 10) updates the ICRE list to take into account the loss of edge $ICRE_i[j]$. To do so, $EndGate$ messages are propagated on the structure tree, and a call to $Prune$ suppresses all unreachable edges in the ICRE list.

Algorithm 10 UpdateGate(j,n)

1: **for all** $k \neq e$ s.t. $Parent_i[k] = i \vee Parent_i[i] = k \vee (i,k) \in ICRE_i$ **do**
2: {for all neighbors in the structure tree}
3: Send *EndGate*$[ICRE_i[j], n]$ to k
4: **end for**
5: $(P_i, ICRE_i) \leftarrow Prune(ICRE_i[j], n)$

e) Algorithm 11: On receiving an *EndGate*$[ICRE, n]$ message: When receiving an $EndGate$ message, if the missing edge appears in the node ICRE list, the node updates its list to take into account the loss of this edge.

Algorithm 11 On reception of *EndGate*$[ICRE, n]$ message

1: **if** $(\exists j / ICRE_i[j] = ICRE)$ **then**
2: UpdateGate(j,n)
3: **end if**

2.5 Merge process

f) Algorithm 12: Initiation of a Merge process: When a node in a structure receives a token from a structure with a lesser id, the latter is absorbed in the node structure. A wave is propagated on the absorbed structure to add the name of the absorbing cluster as a prefix of all clusters name, and a wave is propagated on the absorbing cluster to add 0 as postfix to its name. The link along which the token was sent is added as an ICRE.

To trigger a Merge process, both structures must have been divided (P_t ,$P_i \neq 1$). Else, if the cluster t is smaller than K or his structure has not yet been divided, it should be deleted; to avoid mutual deletion, we use the id of the node that initiated the cluster to break symmetry.

Algorithm 12 On reception of $Token[P_t, Parent_t, ICRE_t]$ message with $P_i \neq 0$ and $ICRE_t[0] \neq ICRE_i[0]$

1: **if** P_i, $P_t \neq 1$ and $|Parent_t| > K$ **then**
2: **if** $(ICRE_t[0] < ICRE_i[0])$ **then**
3: {Absorption process.}
4: $ICREtmp \leftarrow ICRE_t$
5: $append((i(P_t.P_i), e), ICREtmp)$
6: $append(ICRE_i[0], ICREtmp)$
7: **for all** $k \neq e$ s.t. $Parent_i[k] = i \vee Parent_i[i] = k \vee (i(P_i), k) \in ICRE_i$ **do**
8: Send *Absorb*$[ICREtmp, P_t]$ to k
9: **end for**
10: $(P_i, ICRE_i) \leftarrow IRCEUpdate(P_t, ICREtmp)$
11: $P_t \leftarrow P_t.0$
12: $append((e(P_t.0), i), ICRE_t)$
13: Send $Changes[Parent_t, (e(P_t.0), i)]$ to e
14: **end if**
15: Send $Token[P_t, Parent_t, ICRE_t]$ to e
16: **else if** $|Parent_t| < K$ or ($P_t = 1$ and ($P_i \neq 1$ or $ICRE_i[0] < ICRE_t[0]$)) **then**
17: {Token structure is not large enough to be absorbed}
18: Send *EndCluster*$[e]$ to e
19: **else**
20: {Token structure large enough to be involved in merge processes}
21: Send $Token[P_t, Parent_t, ICRE_t]$ to e
22: **end if**

g) Algorithm 13: Propagation of a merge (absorbing cluster): $Changes$ messages are propagated in the absorbing cluster to make all nodes update their cluster name and take into account the new ICRE.

Algorithm 13 On reception of $Changes[Parent, ICRE]$ message

1: $P_i \leftarrow P_i.0$
2: $append(ICRE, ICRE_i)$
3: **for all** k such that $Parent[k] = i$ **do**
4: {Propagate the changes in cluster.}
5: Send $Changes[Parent, ICRE]$ to k
6: **end for**

h) Algorithm 14: Propagation of a merge (absorbed cluster): $Absorb$ messages are propagated in the absorbed structure so that all nodes in it prepend the identifier of the absorbing cluster to their cluster name. The procedure *Prefix*$(P, ICRE_1, ICRE_2)$ returns a prefix of P of length i such as $ICRE_2[i] = ICRE_1[|ICRE_1| - 1]$. It is used to break symmetry between two absorptions from the same structure.

Algorithm 14 On reception of *Absorb*$[ICRE, P_1]$ message

1: **if** $(\exists y : ICRE_i[y] = ICRE[|ICRE|-1]) \wedge ((ICRE[0] < ICRE_i[0]) \vee ((ICRE[0] = ICRE_i[0]) \wedge (P_1 < Prefix(P_i, ICRE, ICRE_i))))$ **then**
2: {Absorption.}
3: **if** $(y \neq 0)$ **then**
4: {Cancel previous absorption.}
5: **if** $(\exists k \in N_i$ such that$(i(P_i), k) = ICRE_i[y-1])$ **then**
6: Send *EndGate*$[(k, i), k]$ to k
7: **end if**
8: $(P_i, ICRE_i) \leftarrow Prune(ICRE_i[y-1], i)$
9: **end if**
10: **for all** $k \neq e$ such that $Parent_i[k] = i \vee Parent_i[i] = k \vee (i(P_i), k) \in ICRE_i$ **do**
11: Send *Absorb*$[ICRE, P_1]$ to k
12: **end for**
13: $(P_i, ICRE_i) \leftarrow IRCEUpdate(P_1, ICRE)$
14: **end if**

Procedure *Prune*$(Edge, j)$ deletes in $ICRE_i$ all ICREs rooted in the node k such that $ICRE_i[k] = Edge$. When an entry is deleted, the corresponding bit in identifier P_i is deleted. The new identifier P_i and the new structure $ICRE_i$ are returned. j is added in $ICRE_i$ if it is empty. j is the identifier of the node who has detected the missing edge.

For example, in cluster 110110, node 209 detects a missing link with node 40. This link is the edge for the fourth level (a, edge between 11011 and 11010). With its *Token* (or a local copy), node 209 knows a subtree rooted by edge $(209, 40)$. It can delete all unreachable levels. For instance, if

$Parent_{110110}$:

1	4	5	6	11	16	209
4	209	4	5	209	11	209

$ICRE_{110110}$:

$(1_{(110110)},12)$	**$(14_{(1101)},3)$**	$(2_{(110111)},30)$	**$(209_{(110110)},40)$**	$(6_{(110110)},7)$

after the application of $Prune$, we obtain for cluster 1110:

$Parent_{1110}$:

1	4	5	6	11	16	209
4	209	4	5	209	11	209

$ICRE_{1110}$:

$(1_{(1110)},12)$	$(2_{(1111)},30)$	$(6_{(1110)},7)$

Procedure *ICREUpdate*$(P_1, ICRE_1)$ returns the new identifier and the new structure of an absorbed cluster after an absorption.

- All edges $(n_1(x), n_2)$ in $ICRE_i$ are replaced by $(n_1(P_1.x), n_2)$;
- $P_i \leftarrow P_1.P_i$;
- For all $j < |ICRE_1| - 2$ such as $ICRE_1[j] = (n_1(P_1), n_2)$ do $ICRE_1[j] \leftarrow (n_1(P_1.0), n_2)$;
- $ICRE_i \leftarrow ICRE_1 \cup ICRE_i$;

3. Sketch of proof

We give below a sketch of the proof of this algorithm.

3.1 Static network with unique initiator

A detailed proof of this algorithm in a static network with a unique initiator can be found in [7]. Below are given the lemmas that structure this proof. The following lemmas hold if no topological change occur.

Lemma 1: There is exactly one token in each elementary cluster.

Lemma 2: Once a node is in a cluster, it remains in it (possibly in its sub-clusters).

Lemma 3: $Child$ messages circulate only between nodes that are part of the cluster being divided. A $Token$ message either comes from a node in the $Token$'s cluster or is bounced back to it.

Lemma 4: Clusters are connected.

Lemma 5: Given a $Token$ message t, $Parent_t$ contains a spanning tree of the elementary cluster t rooted in the last node that held the token.

Lemma 6: If a division process is initiated, based on a broadcast on a tree, the cluster is eventually divided into two sub-clusters.

Lemma 7: Clusters are eventually steady.

Lemma 8: Eventually, all clusters are of size greater than K (provided that there are more than K nodes), and no cluster is divisible.

3.2 Topological changes

Lemma 9: The set of all edges (i, j) such that there is a token with $Parent_t[i] = j$ or $Parent_t[j] = i$ or $(i, j) \in ICRE_t$ is a spanning tree of the visited nodes.

In the sequel, we call structure the set of nodes that have been visited by a token or its descendants.

We now consider the various topological changes that may occur at any point during the execution of the algorithm.

Lemma 10: The apparition of a link has no effect on the algorithm, nor on the specification.

Lemma 11: The disappearance of a link that is not part of the tree described in lemma 9 has no effect on the algorithm, nor on the specification.

Lemma 12: The apparition of a node leads to a configuration corresponding to a possible execution of the algorithm.

Indeed, the same execution of the algorithm is possible on the graph with the extra node.

Lemma 13: After a link in a local tree has disappeared, nodes in the tree below this link are unclustered by algorithms 6, 8 and 9. Eventually, the $Parent$ variable of the $Token$ message (or of the $Token$ messages of children clusters) are corrected to be consistent with the current topology of the system. If the cluster does not reach a size over K, it is eventually deleted.

A node disappearing has the same effect as all its adjacent links disappearing (in particular, those that are part of the tree).

Lemma 14: If a cluster is deleted, its sibling takes the name of its father.

Lemma 15: After a topological change breaking the global tree into two, clusters are renamed consistently with the new global trees.

Lemma 16: After a topological change, the configuration eventually corresponds to a configuration resulting from the algorithm executed on a static network (possibly with multiple initiators).

3.3 Multiple initiators

Lemma 17: If several structures exist, eventually, a token from a weaker structure hits a stronger one (*"weaker"* meaning that the id of the top ICRE is smaller). The weaker structure then triggers its absorption into the other.

Lemma 18: A wave on the global tree of the absorbed structure and a wave on the local tree of the absorbing structure leads to a renaming consistent with a unique structure.

Lemma 17 does not exclude that several absorption are triggered simultaneously.

Lemma 19: In the case of several absorptions triggered simultaneously, only one is carried out.

4. Complexity

All elementary clusters have a size greater than K. Thus, there are less than n/K elementary clusters. Thus, the binary tree of clusters has n/K leaves: since it is a binary tree, its height is between $\log_2(n/K)+1$ and n/K. The building of clusters of disjoint subtrees is parallel. So the time complexity of this algorithm is the height of the clusters tree times the complexity of the building of a divisible cluster starting from a newly divided cluster.

A cluster resulting from a division has at least size K. Now, assume all connected subgraphs of size M in G are divisible. Then, if the cluster reaches size M, it is divisible, and will be divided as soon as its cluster tree is divisible itself. To reach size M (if $M-K$ unclustered nodes exist in its neighborhood), it takes less than the time for a random walk to cover M nodes, the expectation of which is between $O(M \log M)$ and $O(M^3)$.

If the tree is balanced, which should be the case for instance in random graphs, then the height of the tree is less than $\log_2(n/K) = O(\log_2(n))$. In this case, the complexity of the algorithm is $O(\log_2(n))$.

The reactions to topological changes, as well as the merging of different structures, imply waves on the local and global trees. If the global tree is balanced, the extra cost is less than $O(\log_2(n))$. These processes are transparent to a message routed in a sub-structure that is not affected by a topological change.

5. Conclusion

The top-down approach used in this algorithm allows to design a lightweight routing scheme based on the naming scheme. It also lets us devise an approximation of bounded-size clusters.

However, the management of multiple initiators requires then some extra procedure, that can be carried out transparently and efficiently thanks to spanning trees built in the course of the algorithm.

The main step to self-stabilization is to guarantee the existence and the unicity of the token of each cluster, thanks to a process like the one in [6], adapted to the case of random walks in clusters.

References

[1] R. Aleliunas, R. Karp, R. Lipton, L. Lovasz, and C. Rackoff. Random walks, universal traversal sequences and the complexity of maze problems. In *FOCS 79*, pp 218–223, 1979.

[2] David J. Aldous. The random walk construction of uniform spanning trees and uniform labelled trees. *SIAM J. Discret. Math.*, 3: pages 450–465, 1990.

[3] A.D. Amis, R. Prakash, T.H.P. Vuong and D.T. Huynh. Max-min d-cluster formation in wireless ad hoc networks. In *IEEE INFOCOM*, pages 32-41, 2000.

[4] S. Basagni. Distributed clustering for ad hoc networks. In *International Symposium on parallel Architectures, Algorithms and Networks (ISPAN)*, pages 310–315, 1999.

[5] T. Bernard, A. Bui, L. Pilard and D. Sohier. Distributed Clustering Algorithm for Large-Scale Dynamic Networks. *International Journal of Cluster Computing*, Springer eds. DOI: 10.1007/s10586-011-0153-z. 2010.

[6] Thibault Bernard, Alain Bui, Devan Sohier Token Loss Detection for random walk based algorithms. *Seventh IEEE Internationnal Symposium on Parallel and Distributed Computing, vol , pp 351-356, IEEE Computer Society Press, july 2008.*

[7] A. Bui, S. Clavière and D. Sohier. Distributed Construction of Nested Clusters with Inter-cluster Routing In research report, preprint (http://www.prism.uvsq.fr/∼sic/rrBCS12.pdf), 2012.

[8] A. Bui, S. Clavière, A. K. Datta, L. L. Larmore and D. Sohier Self-Stabilizing Construction of Bounded Size Clusters *8th International Colloquium on Structural Information and Communication Complexity SIROCCO 2011*, Lecture Notes in Computer Science, Springer, 2011.

[9] A. Bui, A. Kudireti and D. Sohier. A fully distributed clustering algorithm based on random walks. In *International Symposium on Parallel and Distributed Computing, (ISPDC)*, pages 125–128, 2009.

[10] A. K. Datta, L. L. Larmore and P. Vemula. A self-stabilizing $O(k)$-time k-clustering algorithm. *The Computer Journal*, 53(3): pages 342–350, 2010.

[11] S. Dolev and N. Tzachar. Empire of colonies : Self-stabilizing and self-organizing distributing algorithm. *Theoretical Computer Science*, 410: pages 514–532, 2009.

[12] A. Ephremides, J.E. Wieselthier and D.J. Baker. A design concept for reliable mobile radio networks with frequency hopping signaling. *Proceedings of the IEEE*, pages 56-73, 1987.

[13] C. Johnen and L. Nguyen. Robust self-stabilizing weight-based clustering algorithm. *Theoretical Computer Science* , 410(6-7): pages 581–594, 2009.

[14] L. Lovasz. *Random walks on graphs: A survey.* In T. Szonyi ed. D. Miklos, V. T. Sos, editor, Combinatorics: Paul Erdos is Eighty (vol. 2), p.353–398. Janos Bolyai Mathematical Society, 1993.

[15] J. Sucec and I. Marsic. Location management handoff overhead in hierarchically organized mobile ad hoc networks. In *International Parallel and Distributed Processing Symposium*, (IPDPS), pages 198–, 2002 2:0194, 2002.

[16] D.G. Thaler and C.V. Ravishankar. Distributed top-down hierarchy construction. In *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, pages 693 -701 vol.2, 1998.

[17] S-J. Yang and H-C. Chou. Design Issues and Performance Analysis of Location-Aided Hierarchical Cluster Routing on the MANET In *Communications and Mobile Computing*, (CMC), pages 26–31, 2009.

FPGA Based Physically Unclonable Functions and Neural Networks for Preventing Counterfeiting Problems

Swetha Pappala[1], Mohammed Niamat[1], and Weiqing Sun[2]
[1]Electrical Engineering and Computer Science, University of Toledo, Toledo, Ohio, USA
[2]Engineering Technology, University of Toledo, Toledo, Ohio, USA

Abstract - *Anti-counterfeiting technology/methods have entered a new era with the implementation of critical designs and confidential information transfer protocols. Manufacturing, assembly, and testing are now moving globally outside the company's facility to be done at contract manufacturers making the security of the critical design and information a top priority. According to the Anti-Counterfeiting Bureau, USA, counterfeiting and piracy costs the economy a loss of more than $250 billion in revenue and 750,000 jobs every year. To counter such threats, methodologies have been developed that require a unique signature key for every fabricated chip. Physically Unclonable Functions (PUFs) can be used for such signature generation. This paper presents a design for cryptographic applications that can be implemented on an FPGA taking advantage of its unique architecture. The first part of the research involves techniques for the generation of uniquely distinguishable responses from Ring Oscillator PUFs (ROPUFs) and the latter part involves error correction techniques using Artificial Neural Networks. The proposed design is implemented on several identical Xilinx Spartan FPGAs and the Hamming distances for the responses are computed and analyzed. The uniqueness of the responses is found to be 49.0625%. The results of the proposed Error Correcting Code also prove to be computationally better than the conventional BCH codes.*

Keywords: FPGA, PUF, Security, Error Correcting Code, Neural Network, Bidirectional Associative Memory.

1 Introduction

The complexity in developing security mechanisms and routing protocols for embedded systems continues to increase; on the other hand, cost and size constraints have been lowered. In this scenario, trustworthy authentication of a device is of extreme importance for secure protocols. The continuous increase in density and capability of FPGAs are motivating designers to implement valuable designs using FPGAs. With FPGAs being used in more and more applications that implement valuable designs, significant security features are required. Traditional methods of storing the identity of an object using non-volatile memory are insecure [1]. Hence, a concept of generating a unique digital signature has a wide range of applications in embedded systems security and IC piracy. Physically Unclonable Functions (PUFs) can be used as novel chip identifiers. PUF is a function which produces secret output response based on the underlying properties of a physical device while adhering to various properties based on the deployment and the level of security intended from the device.

PUFs exploit manufacturing process variations in a die to generate non-volatile chip unique signatures exploiting manufacturing process variations of FPGAs. They derive confidential information from uncontrollable random components rather than storing them in the memory. Lack of control over sub-micron process variation makes a PUF unclonable. This enables chip authentication and cryptographic key generation. These process variations are mainly due to the inability to precisely control the diffusion of dopants and due to the inability to robustly fabricate geometric features [1]. These variations result in key electrical parameters of circuit devices and interconnects that increase the uncertainty in the outcome of the design process. PUFs exploit these uncertainties to generate a device specific key.

The quality factor of a PUF, which includes uniqueness, reliability and attack resiliency, is negatively affected by factors like voltage fluctuations and environmental temperature [1]. These factors degrade the stability of the PUF signatures. Hence, an error correction method is necessary to stabilize the circuit. Conventionally, BCH (Bose-Chaudhuri-Hocquenghem) have been used for the error correction process. In this work, Artificial Neural Networks (ANNs) have been used. ANNs are a relatively new signal processing technology. They have a remarkable ability to learn and derive information from complicated and imprecise data that can be used to extract patterns. This characteristic property of ANN gives an advantage over the conventional error correcting codes such as BCH codes [2, 3].

Section II discusses the basic operation of the proposed ROPUFs. Section III discusses the concepts of supervised learning in neural networks and the application of these algorithms to correct the flipped bits in the PUF responses. Section IV concludes the paper.

2 Physically Unclonable Functions

There have been various implementations of PUFs and several sources of variation in silicon PUF. Silicon PUFs can be broadly classified as Memory-based PUFs [4], Delay-based PUFs (RO PUFs) [5, 6] and Glitch Count-based PUFs [7]. The RO PUFs have certain advantages over the other PUFs when implemented on an FPGA platform. Firstly, the implementations of ring oscillators are simplified and can be optimized while routing. Secondly, it has been experimentally observed that under a wide range of temperature fluctuations, RO PUFs are more stable and reliable [2]. Considering the above advantages, RO PUFs have been chosen for the key generation process followed by an error correction technique.

Ring Oscillator PUFs (ROPUFs) are based on variations in frequencies of identical oscillators to produce a secret PUF response. Figure 1 shows the basic principle of RO PUFs. Traditionally, RO PUFs exploit the fact that uncontrollable wire delays and voltage transfer characteristics due to fabrication process variations of FPGA devices cause random but static variations in the frequency of identically laid out oscillators. The response vector, in such PUFs, is generated by a pair-wise comparison of the ring oscillator frequencies. These comparisons are represented as the challenge-response pairs, where the chosen ring oscillator pair is the challenge and the output of the comparator is the response. For an RO PUF with N K-stage identical ring oscillators, there are *N*(N-1)/2* challenge-response comparisons [8]. The oscillators are designed to have identical layouts to ensure that the frequency variations between them are strictly due to process variations.

In this research, we propose a design to implement the RO PUFs on FPGAs using LUTs and multiplexers that are the basic components of FPGA architectures. Figure 2 shows an oscillator of the proposed RO PUF design. Multiple instances are instantiated to generate a multi-bit PUF signature. Two LUTs (LUT X and LUT Y) in a Xilinx Spartan 2 XC2S100 FPGA are used in shift register mode. The shift register contents are initialized to complementary values. As indicated in the figure, the LUTs are initialized to $(0x5555)_{16}$ and $(0xAAAA)_{16}$ respectively. The outputs of the LUTs drive the select input pins of a chain of multiplexers as

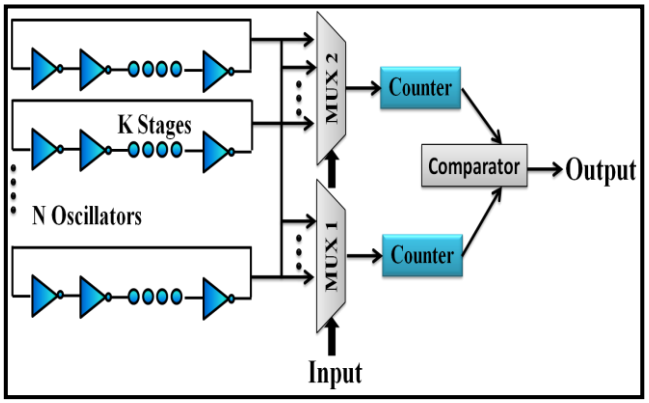


Figure 1. Basic Principle of ROPUFs.

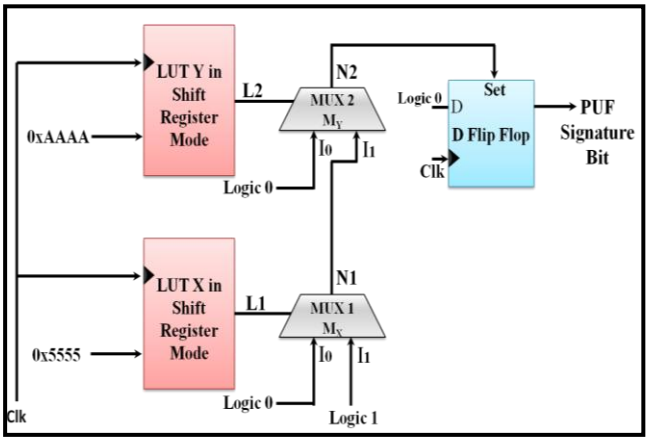


Figure 2. Oscillator Circuit based on the Proposed PUF Circuit.

shown in Figure 2. It is to be noted that both the multiplexers have their I_0 data input tied to logic '0'. The I_1 data line for the bottom multiplexer is tied to logic '1'. On the other hand, the I_1 data line of the top multiplexer is driven by the output of the multiplexer below. The output of the last multiplexer is connected to a D-flip flip which increases the width of the pulse and produces an appropriate output response. The flip flop is initialized to logic '0'.

Due to the complementary initialization values, the shift register implemented in LUT X produces a sequence complementary to LUT Y. Initially, the output of LUT X is logic '0' and the output of LUT Y is logic '1', i.e., signal L1 is logic '0' and signal L2 is logic '1'. Thus, the signals N1 and N2 are both at logic '0'. At the triggered clock edge, the output of LUT X changes from logic '0' to logic '1' and the output of LUT Y changes from logic '1' to logic '0'. Although LUT X and the multiplexer it drives (M_X) are identical to LUT Y and its corresponding multiplexer (M_Y), the two circuits experience different delays due to random process variations. There are two cases worth highlighting. They are:

Case 1: LUT X and the multiplexer it drives are faster than LUT Y and its corresponding multiplexer. In this case, when LUT X transitions from logic '1' to logic '0', the slower LUT Y changes from logic '0' to logic '1', i.e., L1 transitions from logic '1' to logic '0' before L2 transitions from logic '0' to logic '1'. When the signal L1 transitions from logic '1' to logic '0' the signal N2 is held at logic '0'. Thus, the signal N2 is held constant throughout the process.

Case 2: LUT Y and the multiplexer it drives are faster than LUT X and its corresponding multiplexer. In this case, LUT Y transitions from logic '0' to logic '1'when the slower LUT X changes from logic '1' to logic '0', i.e., L2 transitions from logic '0' to logic '1' before L1 transitions from logic '1' to logic '0'. Due to the difference in the delay periods, when L2 has transitioned to logic '1' but L1 is still at logic '1', the select lines of both the multiplexers are held at logic '1' due

TABLE I. SIGNAL TRANSITIONS OF THE PUF.

	L1	L2	N1	N2	Output
Initially	1	0	1	0	0
Case 1 (LUT X faster than LUT Y)	1→0	0	0	0	0
Case 2 (LUT Y faster than LUT X)	1	0→1	1	1(glitch)	1

to which a short positive pulse (glitch) appears in the signal N2. The presence or absence of a glitch on signal N2 and the width of the pulse are due to process variations that impact the relative delays of LUT X and LUT Y. The signal N2 is connected to the SET input line of a D-flip flop as shown in Figure 2. When the glitch appears on the signal N2, the flip flop output and the PUF signature bit becomes logic '1'. In all other cases, the PUF signature bit remains at logic '0'. Table 1 summarizes the operation of the PUF.

Responses of 128-bit length are evaluated using 5 Xilinx Spartan XC2S100 FPGAs at 50MHz. The implementations on each FPGA have eight instances. The Hamming distances for 40 implementations on 5 Spartan 2 FPGAs, i.e., (40*39)/2 = 780 data points are calculated and plotted.

The Hamming distances for intra-chip and inter-chip responses are also analysed. Figure 3 and Figure 4 show the distribution of Hamming distances for the inter-chip and intra-chip 128-bit responses signifying the uniqueness of the PUFs. The figures clearly show the clustering of the distribution around the ranges [61:65] and [56:60]. From the graphs, it has been observed that the uniqueness in the responses is higher in case of inter-chip response bits when compared to the intra-chip response bits. The percentage uniqueness of the PUF responses is calculated to be 49.0625% which is higher than the uniqueness of the responses of a conventional ring oscillator [2] (43.40%) and the chain implementation (48.51%) discussed in [9].

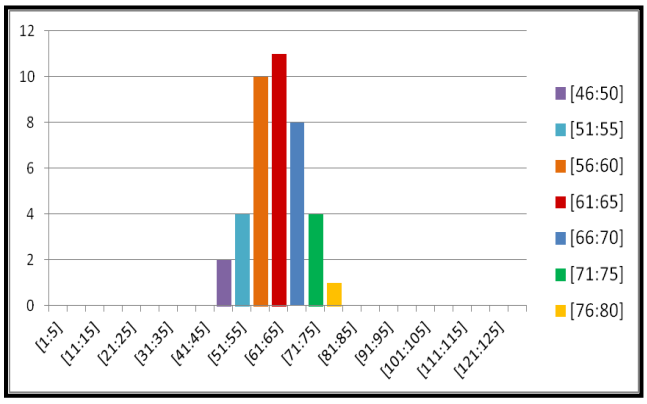


Figure 3. Distribution of Hamming Distances for Inter-Chip Response Bits.

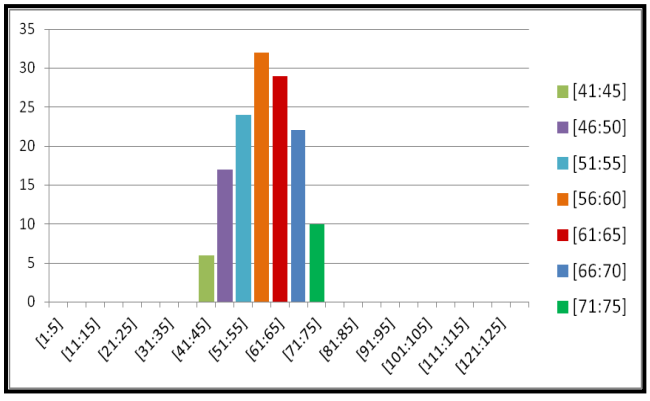


Figure 4. Distribution of Hamming Distances for Intra-Chip Response Bits.

The main advantages of this design are its smaller size and ease of implementation using synthesis, and place and route tools. For example, this design does not need any external routing and the use of hard macros which make them easier to implement.

The stability and reliability of the PUFs are degraded by the slight dependency of the response bits on environmental temperatures and voltage fluctuations. Due to this dependency, a bit generated from a pair of ring oscillators might flip when the operating temperatures change considerably. Thus, the outputs obtained for the same challenge on the same CLB slice may differ slightly. To stabilize the circuit, Artificial Neural Networks (ANNs) are used for its error correction process. The remarkable ability of ANNs to learn and derive information from complicated and imprecise data is used to extract patterns. This gives an advantage over the conventional error correcting codes such as BCH codes [10].

3 Error Correction Process

The BCH codes are a generalization of Hamming codes for multiple error correction. Binary BCH codes were first discovered by A. Hocquenghem, R.C. Bose and D.K. Ray-Chaudhuri. A major disadvantage in case of BCH codes are its complex computations. The computation needed to generate the error correcting syndromes increase drastically with the increase in the number of error bits.

We propose an error correction process using Artificial Neural Networks (ANNs). Our analysis has proved that the learning ability of ANNs has an immense effect on the results produced when compared to the conventional BCH codes. Supervised learning algorithms are effective in dealing with unexpected and changing conditions unlike previously used codes. Bidirectional Associative Memories are used to train the network.

Adaptive Bidirectional Associative Memory (BAM) is a type of recurrent neural network. BAM was proposed by Bart Kosko (1988) [11] as an extension to the Hopfield Network by incorporating an additional layer to perform recurrent auto-associations as well as hetro-associations on the memories. In BAM, bi-directionality, forward and backward

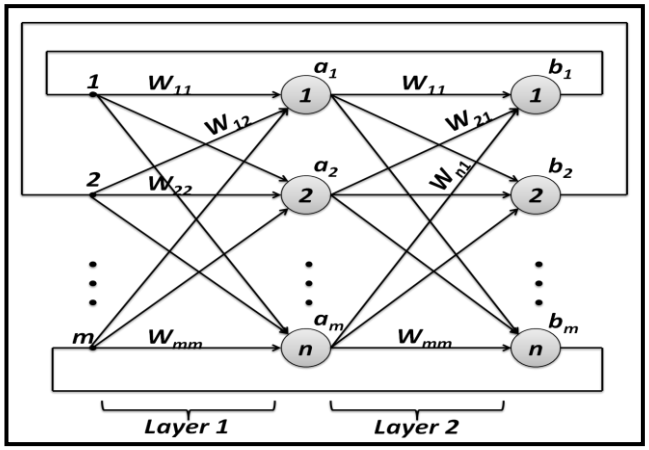


Figure 5. Generalized BAM Architecture.

information flow, is implemented in neural networks to produce two way associative searches for a set of challenge-response associations. The network evolves to a two pattern stable state when the BAM neurons are trained. Hetro-associations are encoded in a BAM by summing correlation matrices. The generalized BAM architecture is shown in Figure 5. The neurons in one layer are fully interconnected to the neurons in the second layer as shown in this figure. There are no interconnections among neurons in the same layer. Layer 1 and 2 operate alternatively. The neuron's output signals are transferred towards the right by using the weight matrix W, and then transferred towards the left by using the transpose of the weight matrix W^T.

In BAM, the correlation matrix for each pattern pair is the matrix product of the transpose of the input vector X^T and the output vector Y. PUF responses are used to train the neural networks. Every weight matrix is bi-directionally stable for bivalent and continuous neurons. The associative weight matrix has been calculated as the sum of all correlation matrices. The mathematical representation is shown in (1).

$$W = \sum_{m=1}^{M} {X_m}^T . Y_m \qquad (1)$$

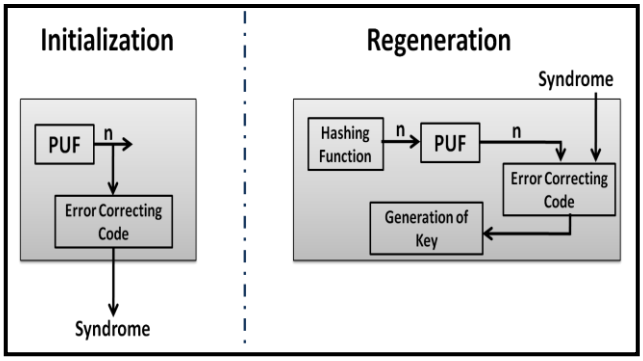


Figure 6. Different Phases of Error Correction Process.

where M is the number of pattern pairs stored in the BAM, X is the input vector and Y is the output vector.

The error correction process has two phases: Initialization Phase and Regeneration Phase. In the initialization phase, multiple outputs are generated from the PUF circuit and an error correcting syndrome is computed by training the neural network. In the regeneration phase, the noisy/corrupt signal is sent to the trained neural network for the errors to be rectified. Figure 6 shows the two phases of an error correction process.

3.1 Training Algorithm

3.1.1 Initialization Phase

To test and analyze the error correcting code, PUF responses are converted into the bipolar binary format. The values are stored in matrix form. The training phase of the neural network generates the weight matrix (error correction syndrome). This is the initialization phase of the error correction process.

3.1.2 Regeneration Phase

In the regeneration phase, a noisy/corrupt signal is sent to the trained neural network for the errors to be rectified. The testing part of this phase has two tasks. Firstly, to ensure no false negatives, the BAM is tested to retrieve any vector from the input matrix. To verify that the network is capable of correctly retrieving any vector from the input matrix, a vector from the input matrix is applied to the neural network. The vector is retrieved using (2).

$$Y_m = sign\ (W^T . X_m)\ where\ m = 1, 2, 3 \ldots . M \qquad (2)$$

The trained vectors are successfully retrieved eliminating false negatives. Secondly, the network should be able to retrieve exact values at the output layer even when a noisy/corrupt input signal is given at the input layer. A noisy test vector is given as an input to the trained network and the error correction process is initialized. The output is calculated using (2). The output values are back-propagated and the input is updated using (3) and the process is repeated iteratively till the input and the output vectors attain equilibrium. Equilibrium is attained when the input and the output vectors remain unchanged with further iterations.

$$X_m = sign\ (W . Y_m) where\ m = 1, 2, 3 \ldots . M \qquad (3)$$

Trained networks have successfully corrected faulty test vectors. In this work, 64 bit, 128 bit and 256 bit vectors are used to test the networks. In the error correcting codes, BAM architecture is used to learn continuous mapping and to rapidly extract bivalent associations from several noisy samples. From the results obtained, it has been observed that learning tends to improve with the increase in the sample size.

MATLAB is used for testing and analyzing the error correction method.

The major advantages of this method over BCH codes are its simple computations which take less time and the ability for group processing. The error correcting syndromes are calculated for a set of PUF outputs rather than one single output vector. All the vectors used to train the network can use the same error correcting syndrome to correct the noisy bits.

The proposed error correcting code using bidirectional associative memories has two other advantages. The technique generates a secure syndrome that does not pose any threats to the PUF response bits used to generate the syndrome. The second major advantage is the robust nature of the algorithm. The failure rates of the error correction code can be driven below 1ppm.

4 Summary and Conclusions

This Silicon PUFs, Delay Based PUFs in specific, are novel chip identifiers based on the slight variations due to the manufacturing processes of identically designed devices. On the other hand, supervised learning algorithms are used to train a network according to given specifications. Our research combines the concepts of ROPUFs, designed using a Xilinx FPGA, with artificial neural networks to generate a unique key for cryptographic applications.

The proposed PUF circuit is implemented on 5 Xilinx Spartan 2 XC2S100 FPGAs and an Agilent 16801A logic analyzer is used to obtain the PUF responses. The intra chip and inter chip responses are analyzed and plotted using Hamming codes. From the graphs, it is concluded that the uniqueness in the responses is higher in case of inter-chip responses than the intra-chip responses. The uniqueness of the responses is found to be 49.0625% which is higher when compared to conventional circuit which signifies higher uniqueness of the responses.

The stability of the PUFs needs to be improvised considering the characteristics of the PUFs drastically affect the performance of the security system. The latter part of the research involves the implementation of an error correction technique using Artificial Neural Networks. The networks are tested using the PUF responses. The networks have successfully corrected the error bits. The failure rates of the proposed method are below 1ppm. Future work includes characterizing the stability of PUF response bits across a wide range of environmental changes and improving the learning rate of the error correcting code to decrease the failure rate.

5 Acknowledgment

This research was partially supported by NSF grant # 0958298.

6 References

[1] A. Maiti, V. Gunreddy and P. Schaumont, "A Systematic Method to Evaluate and Compare the Performance of Physically Unclonable Functions," IACR, 2011.

[2] G. E. Suh and S. Devadas, "Physical Unclonable Functions for Device Authentication and Secret Key Generation," in ACM/IEEE Proceedings of Design Automation Conference, June 2007, pp 9-14.

[3] M. Yu, M. D. Raihi, R. Sowell, S. Devadas, "Lightweight and Secure PUF Key Storage Using Limits of Machine Programming," Workshop on Cryptographic Hardware and Embedded Systems, 2011.

[4] V. van der Leest, G. J. Schrijen, H. Handschuh, and P. Tuyls, "Hardware Intrinsic Security from D flip-flop," ACM Workshop on Scalable Trusted Computing (STC), 2010, pp. 53-62.

[5] X. Xin, J. Kaps, and K. Gaj, "A Configurable Ring-Oscillator Based for Xilinx FPGAs," 14th Euromicro Conference on Digital System Design (DSD), 2011, pp. 651-657.

[6] J. H. Anderson, "A PUF Design for Secure FPGA-Based Embedded Systems," IEEE/ACM Asia and South Pacific Design Automation Conference (ASP – DAC), 2010, pp. 1-6.

[7] D. Suzuki and K. Shimizu, "The Glitch PUF: A New Delay-PUF Architecture Exploiting Glitch Shapes," Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems (CHES), 2010, pp. 366–382.

[8] Dominik Merli, Frederic Stumpf, Claudia Eckert, "Improving the Quality of Ring Oscillator PUFs on FPGAs," Proceedings of the 5th Workshop on Embedded Systems Security, WESS'10, October 2010.

[9] Dominik Merli, Frederic Stumpf, Claudia Eckert, "Improving the Quality of Ring Oscillator PUFs on FPGAs," Proceedings of the 5th Workshop on Embedded Systems Security, WESS'10, October 2010.

[10] M. Yu, M. D. Raihi, R. Sowell, S. Devadas, "Lightweight and Secure PUF Key Storage Using Limits of Machine Programming," Workshop on Cryptographic Hardware and Embedded Systems, 2011.

[11] Bart Kosko, "Bidirectional Associative Memory," IEEE Transactions on Systems, Man, and Cybernetics, vol. 18 no. 1, January/February 1988.

A GPGPU Implementation of Approximate String Matching with Regular Expression Operators and Comparison with Its FPGA Implementation

Yuichiro Utan, Masato Inagi, Shin'ichi Wakabayashi, and Shinobu Nagayama
Graduate School of Information Sciences, Hiroshima City University, Hiroshima, Japan

Abstract— *In this paper, we propose an efficient GPGPU implementation of an algorithm for approximate string matching with regular expression operators, originally implemented on an FPGA, and compare the GPGPU, FPGA and CPU implementations by experiments. Approximate string matching with regular expression operators is used in various applications, such as full text database search and DNA sequence analysis. To efficiently handle a long text in the matching, a hardware algorithm for FPGA implementation has been proposed. However, due to the limitation of FPGAs' capacity, it cannot handle long patterns. In contrast, our proposed GPGPU implementation is able to handle long patterns efficiently, utilizing the scalability of GPGPU programming. Experimental results showed that the GPU implementation is more than 18 times as fast as the CPU one when the pattern length is greater than 3200, while the FPGA one could not handle such a long pattern.*

Keywords: approximate string matching, regular expression, GPGPU, FPGA, CUDA

1. Introduction

Approximate string matching [1] is the problem to find substrings in a given string (*text*) which are similar to another given string, called a *pattern*. The degree of similarity between two strings, called *edit distance*, is obtained by deriving the *edit distance matrix*. Approximate string matching is one of the major problems in information science, and used in keyword search in databases, DNA sequence analysis in bioinformatics, and network intrusion detection, *etc*. Since the problem size is exponentially increasing, some efficient algorithms using special hardware [2] or a graphics processing unit (GPU) [3] have been proposed.

Although approximate string matching is more flexible than exact string matching, both kinds of matching deal with only simple patterns which consist of only alphabet characters. Such simple descriptions lead unacceptably long patterns in some applications in which the target substrings vary under certain rules (*e.g.*, network intrusion detection). Thus, matching that can deal with more efficiently described patterns is being required. One of the methods for efficiently describing patterns is regular expression. In this paper, we call a variant of approximate string matching in which patterns can include some regular expression operators, *approximate regular expression matching*. A systolic hardware algorithm for approximate regular expression matching has already been proposed and implemented on an FPGA in [4]. However, in this algorithm, the acceptable pattern length is limited by the amount of hardware resources (*i.e.*, the FPGA's capacity), and thus only short patterns can be handled.

Recently, parallel processing methods using GPUs are attracting attention. GPUs are originally application specific processors for graphics processing. However, since GPUs have highly parallel architectures to render pixel images in real-time, general-purpose computation on GPUs (GPGPU) [5] has been actively being studied to utilize GPUs' high performance. As a development environment for GPGPU, compute unified device architecture (CUDA) is provided by NVIDIA Corporation to facilitate the utilization of GPUs for general-purpose computing. GPGPU is based on homogeneous multithreading, and threads more than hardware resources (*e.g.*, ALUs) are automatically scheduled and allocated to the resources. Thus, GPGPU programming is scalable in terms of the number of threads.

Utilizing the high parallelism of GPUs, a GPGPU implementation of approximate string matching has been proposed in [3]. This method enhances the degree of data parallelism by handling multiple texts in parallel. Thus, it cannot fully utilize a GPU's parallelism when performing matching with a single text and a long pattern.

In this paper, we propose an efficient GPGPU implementation of approximate regular expression matching for long patterns. Our method is based on the FPGA implementation [4]. Comparing to the FPGA implementation, the main advantages of the GPGPU implementation are:

1) our method can handle much longer patterns, and
2) it requires no special hardware, like FPGAs.

The main differences of our proposed method from [3] are:

1) it can handle some regular expression operators, and
2) it is suitable to high-speed matching with a single text.

The method proposed in [3] divides the edit distance matrix into parallelogram regions. In the method, when only a single text is given, up to 32 elements in a region are calculated in parallel, and the regions are sequentially handled. Our

method divides the matrix into parallelogram regions in the same way. In our method, however, up to 32 elements in a region are calculated in parallel, and up to 16 regions are calculated in parallel, considering the data dependencies among regions. (The degrees of parallelism depend on the GPU.) This makes our method more efficient for matching with a single text than [3].

In addition, we evaluate the GPGPU implementation comparing to the FPGA implementation by experiments. The experimental results showed that the FPGA and GPU implementations were 8.3 and 2.9 times as fast as a CPU implementation when the pattern length is 320, respectively. Furthermore, the GPU implementation was more than 18 times as fast as the CPU one when the pattern length is greater than 3200, while the FPGA one could not handle such a long pattern.

The rest of this paper is organized as follows. In Section 2, the definitions of approximate string matching and approximate regular expression matching are described, and GPGPU is explained. Section 3 shortly describes the existing FPGA implementation of approximate regular expression matching [4]. Section 4 presents our GPGPU implementation of approximate regular expression matching. In Section 5, we compare the GPGPU, FPGA and CPU implementations by experiments. Finally, conclusions are given in Section 6.

2. Preliminaries

2.1 Approximate String Matching

Here, we define the approximate string matching problem and explain an algorithm for the problem. Given two strings P (pattern) and T (text), and a non-zero integer k (threshold), the approximate string matching problem is to find a substring of T whose edit distance [6] from P is less than or equal to k. Now, let us consider transforming a string S_1 to another string S_2 by iteratively applying single-character deletions, insertions and substitutions. When the costs of deletions, insertions and substitutions are given, the edit distance between S_1 and S_2 is the minimum total cost required to transform S_1 to S_2. Thus, the calculation of edit distance is essential to the approximate string matching problem.

Next, we explain how to calculate the edit distance. The edit distance between S_1 and S_2 is calculated as $D(m,n)$ defined in the following by using dynamic programming (DP), where $m = |S_1|$ and $n = |S_2|$.

$$D(i,j) = \min\{ \begin{array}{l} D(i-1,j) + \text{del}, \\ D(i,j-1) + \text{ins}, \\ D(i-1,j-1) + s(i,j) \end{array} \}, \quad (1)$$

where

$$s(i,j) = \begin{cases} \text{sub}(S_1[i], S_2[j]) & 1 \le i \le m, 1 \le j \le n \\ \infty & \text{otherwise,} \end{cases}$$

$S_1[i]$ and $S_2[j]$ are the i-th character of S_1 and the j-th character of S_2, respectively, $D(0,0) = 0$, and $D(i,j) = \infty$ if $i < 0$ or $j < 0$. ins and del in the formula are constants and denote the insertion and deletion costs, respectively. $\text{sub}(a,b)$ is the substitution cost of two characters a and b. sub is represented as an $\alpha \times \alpha$ two-dimensional array, where α is the number of alphabet characters. For discussion, let D be an $(m+1) \times (n+1)$ two-dimensional array such that $\text{D}[i,j] = D(i,j)$. D is called the edit distance matrix.

2.2 Approximate Regular Expression Matching

Here, we explain how to introduce regular expression operators and other operators into approximate string matching. Hereinafter, "a string P matches a string S" means that the edit distance between P and S is zero. The definitions of the main target operators are as follows.

1) *Single-character don't care* (*SCDC*) (?)
   The pattern "?" matches any single character.
2) *Variable-length don't care* (*VLDC*) (?*)
   The pattern "?*" matches any string, including the zero-length string "ε", where ε is the empty character.
3) *Negation* ($\bar{p}$)
   A pattern "$\bar{p}$" matches any single character other than p.
4) *Empty character matching* ($p@$)
   A pattern "$p@$" matches p and the empty character ε.
5) *Character-by-character matching* (*CC matching*) ($[p_1p_2\cdots p_l]$)
   A pattern $P =$ "$[p_1p_2\cdots p_l]$" matches only the string "$p_1p_2\cdots p_l$". If P and S have different lengths, the edit distance between P and S is ∞. Otherwise, the edit distance between P and S is the same as that between "$p_1p_2\cdots p_l$" and S.
6) *Exact matching* ($\langle[p_1p_2\cdots p_l]\rangle$)
   A pattern $P =$ "$\langle[p_1p_2\cdots p_l]\rangle$" matches only the string "$p_1p_2\cdots p_l$". If S is not "$p_1p_2\cdots p_l$", the edit distance between P and S is ∞.
7) *Kleene operator* (p^*)
   A pattern "p^*" matches any strings that do not include any character other than p.

The other target operators are shown in [4].

Due to space limitations, we here explain only the DP formulation of exact matching. Assume $P = \langle[p_1p_2\cdots p_l]\rangle$. Let $\text{ins}(i)$ be the insertion cost of a character between p_i and p_{i+1}. Also, let $\text{sub}(p,t)$ be the substitution cost from a pattern character p to a text character t. Exact matching is realized by setting the deletion, insertion and substitution costs as defined in Expressions (2), (3) and (4), respectively. Note that although insertions right after p_i ($1 \le i < l$) are not allowed, insertion right after p_l is allowed if some pattern characters follow the exact match (*e.g.*, "$\langle[abc]\rangle e$" matches "$abcde$"). Thus, the constant insertion cost $\text{ins}(= k)$ in the original DP formulation is replaced by the function $\text{ins}(i)$.

$$\text{del} = \infty \quad (2)$$

Table 1: Definitions of del, ins, and sub for each operator

operator	del	ins	sub
w/o operator			
p	del	ins	$\mathrm{sub}(p,t)$
1. SCDC			
$?$	del	ins	0
2. VLDC			
$?^*$	0	0	0
3. Negation			
$\bar{p}$	del	ins	$\mathrm{sub}(\bar{p},t)$
4. Empty character			
$p@$	0	ins	$\mathrm{sub}(p,t)$
5. CC matching			
$[p_1p_2\cdots p_l]$			
$1 \le i < l$	∞	∞	$\mathrm{sub}(p,t)$
$i = l$	∞	ins	$\mathrm{sub}(p,t)$
6. Exact matching			
$\langle[p_1p_2\cdots p_l]\rangle$			
$1 \le i < l,\ p = t$	∞	∞	0
$1 \le i < l,\ p \neq t$	∞	∞	∞
$i = l, p = t$	∞	ins	0
$i = l, p \neq t$	∞	ins	∞
7. Kleene operator			
p^*	0	$\min\{\mathrm{ins}, \mathrm{sub}(p,t)\}$	$\mathrm{sub}(p,t)$

$$\mathrm{ins}(i) = \begin{cases} \infty & 1 \le i < l \\ k & i = l \end{cases} \tag{3}$$

$$\mathrm{sub}(p,t) = \begin{cases} 0 & p = t \\ \infty & p \neq t \end{cases} \tag{4}$$

The other operators can be realized by similarly replacing the deletion cost del, insertion cost ins and substitution cost sub. The definitions of del, ins and sub for each operation are shown in Table 1.

2.3 GPU

A graphics processing unit (GPU) is an application specific processor for graphics processing and one of the main components of PCs. Utilizing GPUs for general-purpose computing is called GPGPU (general-purpose computation on GPUs) [5]. In recent years, GPUs without video outputs were developed for general-purpose computing by some semiconductor companies (*e.g.*, Tesla C2070 by NVIDIA Corporation and FireStream 9350 by Advanced Micro Devices, Inc.).

2.3.1 GPU Architecture

Fig.1 illustrates an architecture of NVIDIA's GPUs, called Fermi. It is composed of two LSIs, a GPU itself and a memory chip, called device memory. A GPU has up to 16 streaming multi-processors (SMs), each of which corresponds to a core in a multicore CPU. An SM has 32 streaming processors (SPs), each of which corresponds to an arithmetic logic unit in a CPU core. An SP has only limited functions such as arithmetic operations, and SPs execute instructions decoded by an SM in a SIMD fashion. That is, all the SPs in an SM simultaneously execute the same instruction. In addition, an SM has a memory shared by all the SPs in the SM and some memories to cache data from and to the device memory.

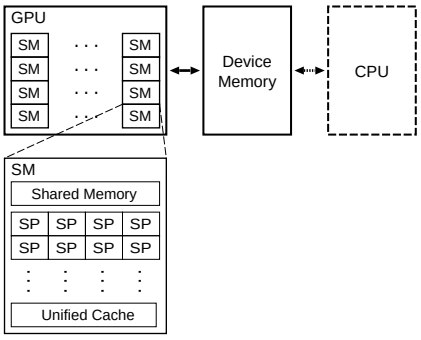


Figure 1: Architecture of NVIDIA Tesla series GPU

GPUs have a hierarchical memory architecture. The memory architecture of the Fermi GPUs is as follows:

1) Global memory
   Global memory is a high-capacity memory realized by the device memory. It is accessible from any SPs in the GPU. The size of the memory is up to several giga-bytes. On the other hand, its latency is high and it requires 400 to 600 clock cycles to access. Accesses from SPs to the memory are cached in the SM to which the SP belongs. It is the only memory readable/writable from both the GPU and CPU.
2) Shared memory
   Each SM has a 64KB on-chip memory, and 16KB or 48KB of the memory is used as its shared memory. The rest of the on-chip memory is used as the L1 cache of global memory. All the SPs in an SM are quickly accessible to the memory.
3) Register
   Each SP has its own registers. Registers are the memory most quickly accessible from SPs. Registers in an SP is not accessible from the other SPs. Each SM has 8K to 32K registers, depending on the GPU.
4) Constant and texture memories
   Constant and texture memories are read-only memories realized by the device memory. They are accessible from any SPs in the GPU. Accesses from an SP to the memories are cached in the SM to which the SP belongs. They are writable from the CPU. In addition, texture memory has some additional functions, such as address space normalization to [0,1] and data interpolation between adjacent data points. Since the proposed method does not use those memories, we omit the detailed explanation of them.

2.3.2 CUDA

CUDA is a programming environment for developing and executing general-purpose applications on NVIDA's GPUs.

It makes multi-thread applications run on GPUs efficiently. An extended C/C++ is used as the programming language in CUDA.

In CUDA, concepts to manage threads, called *grid* and *thread block*, are introduced. A group of threads is called a thread block. The maximum number of threads in a thread block is 512. A group of thread block is called a grid. Each thread block in a grid is managed by adding a two-dimensional ID. The maximum number in each dimension of a thread block ID is 65,535.

Each thread executes a code, called a *kernel*. Each thread has a unique ID, by which threads handle different data, executing the same code. Threads in the same thread block shares data in the high-speed shared memory. On the other hand, a thread cannot access to shared memories in different thread blocks. Thus, codes need to be written so that there are as few data dependencies between thread blocks as possible. In addition, although threads in different thread blocks cannot be synchronized during a kernel execution, threads in the same thread block can be synchronized using a command and keep data consistency.

3. FPGA Implementation of Approximate String Matching with Regular Expression Operators

In this section, a hardware algorithm for approximate regular expression matching proposed in [4] is explained. Our proposed method is based on the same idea and adjusted to GPUs.

3.1 Architecture

The architecture for the hardware algorithm is shown in Fig. 2. Let $m = |P|$ and $n = |T|$. Then, it is a one-dimensional array of $m+1$ units, called *cells*, each of which compares a pattern character and a text character. That is, each cell calculates the elements in a row of the edit distance matrix D, shown in Fig. 3. Note that each dimension of D is expanded by one element in order to calculate the edit distances between the pattern and substrings of the text (please refer to [4] for more detail). The $m+1$ cells calculate the elements on a diagonal line in parallel and the calculation proceeds from the top-left corner to the bottom-right corner as shown in Fig. 3. Thus, its calculation time is $O(m+n)$. The resultant edit distances are output from the right-most cell and input to a comparator. The comparator compares the user-defined threshold k and an edit distance. If the edit distance is less than or equal to k, the comparator outputs a match signal. Since each cell handles one character in a pattern, the length of patterns is limited to the number of cells.

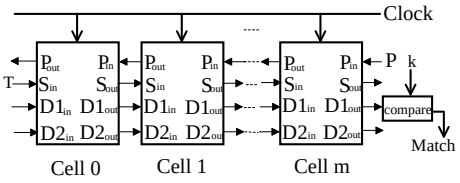


Figure 2: Architecture of Hardware Engine

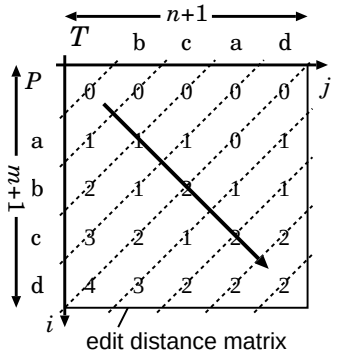


Figure 3: Order of Calculation in D

3.2 Basic Structure and Behavior of Cell

Fig. 4 shows the structure of a cell. C_p and C_t are registers and store one character in the pattern and one character in the text, respectively. DM is a memory to store the table of substitution costs. DM is a two-dimensional array of substitution cost from alphabet α_1 to α_2, called a distance matrix. That is, DM$[\alpha_1, \alpha_2]$ stores the value of sub(α_1, α_2). The distance matrix DM is set before starting matching. (In our experiment, the substitution costs of any two characters are set to 1 and DM is omitted.) $D1$ and $D2$ are registers to temporarily store elements of D. $D1$ stores the newest element the cell calculated, and $D2$ stores the second newest element.

Next, we explain the behavior of the cell. A cell i handles one character of the pattern, p_i, and calculates $D[i, *]$. When the cell i calculates $D[i, j]$ at a clock cycle T_k, $D[i, j-1]$, $D[i-1, j]$ and $D[i-1, j-1]$ are necessary. $D[i, j-1]$ was calculated at the cell i at the clock cycle T_{k-1} and currently stored in $D1$. $D[i-1, j]$ was also calculated at the cell $i-1$ at the clock cycle T_{k-1} and currently stored in $D1$ of the cell $i-1$. $D[i-1, j-1]$ was calculated at the cell $i-1$ at the clock cycle T_{k-2} and currently stored in $D2$ of the cell $i-1$. Thus, all the elements necessary to calculate $D[i, j]$ are stored in the cells i and $i-1$.

4. GPGPU Implementation of Approximate Regular Expression Matching

In this paper, we propose an efficient GPGPU implementation method of the hardware algorithm for approximate

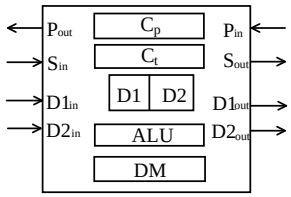


Figure 4: Structure of Cell

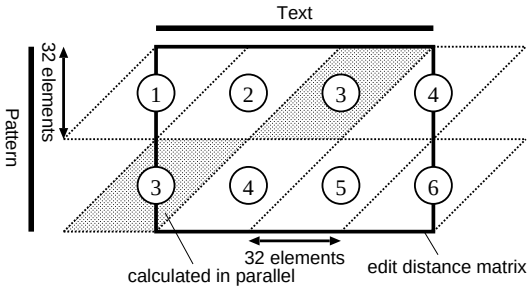


Figure 5: Parallel Calculation on GPU

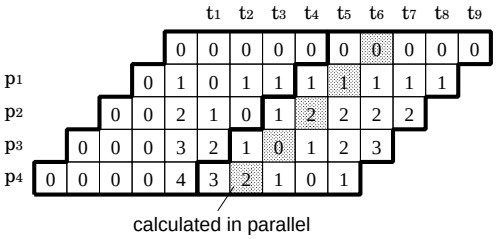


Figure 6: Calculation in Parallelogram Region

regular expression matching [4]. Although GPUs are processors specific to graphics processing, some studies have been conducted to utilize their highly parallel architecture for string matching (*e.g.*, [3]).

In [4], the hardware algorithm is implemented on an FPGA. However, since the length of patterns is limited to the number of cells and the number of cells is limited by the capacity of the FPGA, long patterns cannot be handled by the FPGA implementation. For example, the one implemented in the experiments in [4] can handle no more than 250 pattern characters. In contrast, since in a GPU an SM can handle multiple thread blocks in a time-division manner (*i.e.*, an SP handles multiple threads), a GPU can handle patterns whose length is greater than the number of SPs by allocating the function of each cell to a thread. This fact makes approximate regular expression matching applicable to the applications with long patterns (*e.g.*, analysis of DNA sequences).

In the following, we first show the overview of our method for approximate string matching (without regular expression) on GPUs. Then, we explain the effective memory access method for our matching method. Finally, we introduce regular expression operators to the method.

4.1 Division of Edit Distance Matrix D

In this paper, we propose an efficient approximate regular expression matching method that can handle long patterns utilizing a GPU. In our method, we divide the edit distance matrix D into multiple parts and effectively dispatch them to SMs.

First, we divide the edit distance matrix into the parallelogram regions so that the length of each side of the regions is 32, as shown in Fig. 5. This is because each SM handles 32 threads as one executable unit, called a *warp*. Then, we dispatch the calculations in parallelogram regions to SMs in a GPU. Since SPs in different SMs cannot be synchronized, the parallelogram regions are calculated in the order shown in Fig. 5 to maintain data dependencies. The parallelogram regions with the same number are calculated by multiple SMs in parallel. The calculations of the parallelogram regions with each number are started by calling a kernel and thus synchronized. The elements on a line parallel to the right and left sides of a parallelogram are calculated in parallel by the SPs in a SM. This division realizes an effective use of SMs and SPs in a GPU for the calculation of the edit distance matrix with a long pattern.

Fig. 6 shows the calculation in each parallelogram region. The elements in a region have the same data dependencies as that of Fig. 3. Thus, we allocate the function of a cell to a thread to calculate the elements on a line parallel to the right and left sides of the parallelogram region in parallel.

In summary, in our proposed method, the edit distance matrix is divided into parallelogram regions, and the regions are calculated in the order shown in Fig. 5 in parallel. In each region, the elements on a line parallel to the right and left sides of the parallelogram region are allocated to SPs and calculated in parallel.

4.2 Calculation of Edit Distance on Shared Memory

Here, we describe how to calculate the edit distance using shared memories. The calculation of the edit distance is easily implemented by placing whole the edit distance matrix D to the global memory. However, the cost to access to the global memory is very high and the latency is 400 to 600 clock cycles. Therefore, we utilize shared memories to calculate the matrix.

To calculate the value of elements in a parallelogram region, the calculation results of other regions are necessary. Since different regions are handled by different SMs, it is necessary for SMs to access the global memory to communicate each other. On the other hand, since SMs do not need to communicate each other when calculating inner

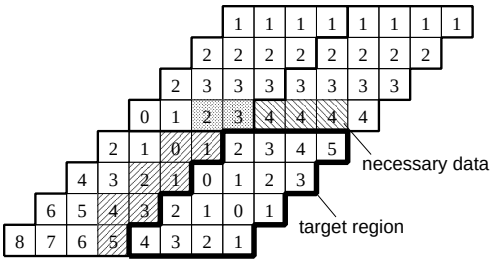


Figure 7: Data Dependencies among Parallelogram Regions

elements of regions, shared memories can be used to quickly calculate the elements. To calculate the elements on the i-th line parallel to the right and left sides of the parallelogram region, only the elements on the $(i-1)$-th and $(i-2)$-th lines need to be stored in the shared memory of the corresponding SM.

The data dependencies among parallelogram regions are shown in Fig. 7. The calculation in the parallelogram region enclosed by a heavy line requires only the shaded elements. Thus, in our method, only those elements are stored in the global memory. In other words, only the elements on the last two lines parallel to the right and left sides of a parallelogram region and those in the bottom row in the region are stored in the global memory.

Fig. 8 shows the pseudo code of the kernel. In the code, *r_D* is a register to temporarily store an element of the edit distance matrix, $D[i, j]$. *r_left*, *r_top*, and *r_diag* are registers to temporarily store the elements $D[i-1, j]$, $D[i, j-1]$, and $D[i-1, j-1]$, respectively. The register *r_left* is used to send the value of $D[i-1, j]$ for the calculation of $D[i+1, j]$ without using shared memories. The registers *r_top* and *r_diag* are used to shorten the *then* and *else* statements in the *if-then-else* statement. This is because a GPU executes both of *then* and *else* statements to execute threads in a SIMD fashion. Without *r_top* and *r_diag*, both of the *then* and *else* statements need to include similar codes (corresponding to the 18th line, which becomes more complicated when regular expression operators are introduced), and it degrades the performance. *idx* is the ID number of the thread in the grid, *tid* is the ID number of the thread block, and *b_id* is the ID number of the thread block. *text*, *top*, and *down* are arrays located in the shared memory to store text characters, the elements on the region, and the elements in the bottom row in the region, respectively.

In the 1st to 7th lines of the code, the data needed by the thread block are read from the global memory to the shared memory. Note that the data are read in parallel by using all the thread in the thread block. In the 8th to 24th lines, the elements in a row are calculated. Note that different threads in a thread block handle different rows, and the elements in the rows are calculated in parallel. In the 9th line, a text character is read to the register *t*. DM in the 10th line is a

```
01.  text[tid] = T[tid+(a-1)*SIZE];
02.  text[SIZE+tid] = T[tid+(a)*SIZE];
03.  top1[tid+1] = D[(b_id)*(SIZE+n+1)+tid+(a+1)*SIZE];
04.  s1[tid] = D[(idx+1)*(SIZE+n+1)+a*SIZE];
05.  s2[tid] = D[(idx+1)*(SIZE+n+1)+a*SIZE-1];
06.  r_E = D[(idx+1)*(SIZE+n+1)+a*SIZE];
07.  p = P[idx];
08.  for(i=0; i<SIZE; i++){
09.      t = text[tid];
10.      sub = DM[p*128+t];
11.      if(tid==0){
12.          r_top = top[i+1];
13.          r_diag = top[i];
14.      }else{
15.         r_top = s1[tid-1];
16.         r_diag = s2[tid-1];
17.      }
18.      r_D=min(r_top+del, r_left+ins, r_diag+sub);
19.      s1[tid] = r_D;
20.      s2[tid] = r_top;
21.      r_left = r_D;
22.      down[i] = r_D;
23.      __syncthreads();
24.  }
25.  D[(idx+1)*(SIZE+n+1)+SIZE+a*SIZE] = r_D;
26.  D[(idx)*(SIZE+n+1)+SIZE-1+a*SIZE] = r_top;
27.  D[(b_id*SIZE)*(SIZE+n+1)+tid+(a*SIZE)] = down[tid];
```

Figure 8: Pseudo Code of Kernel

two-dimensional array to store the substitution costs. In this line, the substitution cost of the pattern character and text character is obtained. In the 11th to 17th lines, $D[i-1, j]$ and $D[i-1, j-1]$ are read from the shared memory to registers. In the 18th line, $D[i, j]$ is calculated. In the 19th to 22nd lines, $D[i, j]$ and $D[i, j-1]$ in the shared memory are updated. Note that only the newest and second newest lines of elements are stored in the shared memory. In the 23rd line, all the threads in the thread block are synchronized (because each SP handles multiple threads). In the 25th to 27th lines, the date necessary to calculate the next regions are sent from the shared memory to the global memory.

4.3 Introduction of Regular Expression Operators

As shown in Section 2, our target regular expression operators can be realized by replacing del, ins and sub in the DP formula of the edit distance calculation. To input patterns with the regular expression operators to the kernel, we introduce an array of characters *OP* ($|OP| = |P|$) for representing operators to the corresponding pattern characters, and an array of integers *e* ($|e| = |P|$) for representing the first and last characters of the target substring of an operator, as input data to the kernel. In the kernel, *OP* and *e* are copied from the global memory to the shared memory like as pattern *P*.

5. Comparison of GPGPU and FPGA Implementations

To compare FPGA, GPGPU and CPU implementations, we conducted some experiments. For those implementations,

we used a PC equipped with an Intel Core i7 950 CPU (3.06GHz), 24GB main memory, CentOS, an NVIDIA Tesla C2070 (1.15GHz) GPU with 4GB device memory, and an FPGA board with a Xilinx Virtex-4 FPGA. The GPU and the FPGA board are connected to the PC with PCIe 2.0 x16 and conventional PCI buses, respectively. The FPGA implementation is written in Verilog HDL and mapped using Xilinx ISE 13.1.

In our GPGPU implementation, one thread block consists of 32 threads. Thus, each thread block uses 1060bytes of the shared memory. Each thread uses 42 registers. Since in the Tesla C2070 each SM has a 48KB shared memory, each SM can handle up to 45 thread blocks ($45 \times 1060 < 48$K). Therefore, our GPGPU implementation can handle long patterns whose length is less than or equal to 20,160 = $14 \times 45 \times 32$ = (the number of SMs) $\times$ (the maximum number of thread blocks per SM) $\times$ (the number of threads in one thread block). In contrast, since only 250 cells can be implemented on the target FPGA, the FPGA implementation can handle only patterns whose length does not exceed 250. The maximum clock frequency was 140MHz.

Table 2 shows the execution times of the GPGPU and CPU implementations when $|T| = 3,200,000$ and $|P| = 320$, and that of the FPGA implementation when $|T| = 3,200,000$ and $|P| = 250$. Note that the execution times include data transfer time from the main memory. As a result, the FPGA and GPGPU implementations were 8.3 and 2.9 times as fast as the CPU implementation, respectively.

Table 3 and Fig. 9 show the results in the cases of long patterns. We found that 1) the execution time of the CPU implementation is proportional to the pattern length, 2) that of the GPGPU implementation is stepwise in terms of the pattern length (there is a gap between $|P| = 3200$ and $|P| = 4800$). In addition, the GPGPU implementation when $|P| \geq 3200$ was more than 18 times as fast as the CPU implementation.

These results indicate that the FPGA implementation is the fastest and suitable to the cases of short patterns, and the GPGPU implementation is scalable and suitable to the cases of long patterns, such as analysis of DNA sequences.

Table 2: Execution Time when $|T| = 3,200,000$

Method	$\|P\|$	Time
CPU	320	23.60 [s]
GPGPU	320	8.23 [s]
FPGA	250	2.89 [s]

Table 3: Execution Time when $|T| = 320,000$ [ms]

$\|P\|$	320	640	1280	3200	4800	6400
CPU	2399	4803	9612	24020	36030	47960
GPU	1079	1102	1131	1212	1922	2031

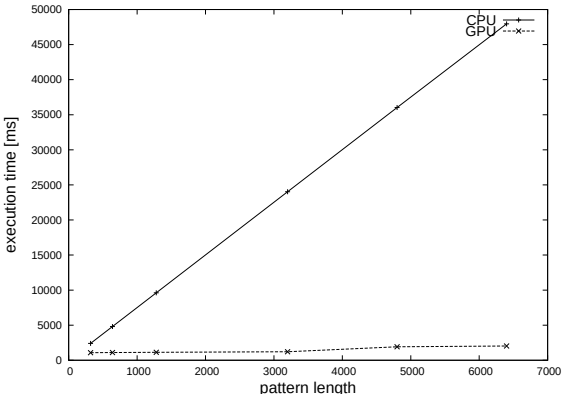


Figure 9: Execution Time of CPU and GPU Implementations

6. Conclusions

In this paper, we proposed an efficient GPU-based method for approximate regular expression matching with long patterns. Experimental results showed that 1) our proposed method and an FPGA-based method [4] are 2.9 and 8.3 times as fast as a CPU implementation, respectively, when the length of patterns is 320; 2) our method is more than 18 times as fast as the CPU implementation when the length of patterns is more than 3200. Our future work includes further improvement of memory access efficiency in our GPU-based method.

References

[1] J. Aoe (Ed.), *Computer Algorithms: String Pattern Matching Strategies*, IEEE Computer Society Press, 1994.

[2] S. Mikami, Y. Kawanaka, S. Wakabayashi, and S. Nagayama, "Efficient FPGA-based hardware algorithms for approximate string matching," in *Proc. the 23rd International Technology Conference on Circuits/Systems, Computers and Communications (ITC-CSCC 2008)*, July 2008, pp.201–204.

[3] K. Dohi, K. Benkridt, C. Ling, T. Hamada, and Y. Shibata, "Highly Efficient Mapping of the Smith-Waterman algorithm on CUDA-compatible GPUs," in *Proc. the 21st IEEE International Conference on Application-specific System Architecture and Processors (ASAP 2010)*, July 2010, pp. 29–36.

[4] Y. Utan, S. Wakabayashi, and S. Nagayama, "An FPGA-Based Text Search Engine for Approximate Regular Expression Matching," in *Proc. the 2010 International Conference on Field-Programmable Technology (FPT'10)*, Dec. 2010, pp. 69–74.

[5] E. Kandrot and J. Sanders, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, July 2010.

[6] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no.1, pp. 168–178, 1974.

Accelerating and Characterizing Seam Carving Using a Heterogeneous CPU-GPU System

Ronald Duarte and Resit Sendag
Department of Electrical, Computer, and Biomedical Engineering,
University of Rhode Island, Kingston, RI, USA

Abstract - *Seam carving has been widely used for content-aware resizing of images and videos with little to no perceptible distortion. Unfortunately, for high-resolution videos and large images it becomes computationally unfeasible to do the resizing in real-time using small-scale CPU systems. In this paper, we exploit the highly parallel computational capabilities of CUDA-enabled Graphics Processing Units (GPUs) for accelerating the content-aware resizing of videos and images. The performance results show that our implementation of the seam carving algorithm achieves up to 100x and 14x speed-ups on the computationally-intensive part of the algorithm compared to the faster single-threaded and the faster multithreaded CPU implementations, respectively, on the systems tested. The overall resizing operation is over 6x and 2x faster than the best single-threaded and multithreaded CPU implementations, respectively, which demonstrates the potential to resize videos and large images in real-time.*

Keywords: Seam carving, GPU, CUDA, parallelization, heterogeneous system

1 Introduction

One of the most popular uses of diverse mobile devices today is for browsing images and playing videos. However, different devices have different resolution capabilities, so it is necessary to resize images and videos efficiently and effectively to fit them into diverse displays (such as cell phones, PDAs, desktop displays, etc), preferably without distortion.

Cropping [1-5] has been one of the most popular approaches to resize images. However, cropping may lose an unacceptable amount of visual information when important structures lie at all edges of an image. In addition, it can only remove information, but it cannot add information to expand the image. Scaling methods, with or without interpolation, tend to produce distorted images, especially when an image is scaled in one direction.

Avidan and Shamir [6] recently provided a new approach to image and video resizing, called *seam carving*. Seam carving functions by establishing a number of seams (paths of least importance) in a digital media and automatically removes or inserts seams to resize the media. This content-aware resizing method has been shown to effectively resize images and videos with little to no perceptible distortion. Seam carving is a computationally-intensive operation. For high-resolution images and videos, it could be difficult to perform the resizing in real-time by using the CPUs in a desktop-scale computer.

The advent of commodity massively parallel architectures, such as modern GPUs, is a compelling option for inexpensively removing the computationally-intensive operations from the CPU. In this paper, we exploit the data-parallel execution model of GPUs for the implementation of content-aware image and video resizing. This paper makes the following contributions:

1) We evaluate GPU-based seam carving algorithm on two CUDA-enabled NVIDIA GPUs.
2) We compare single- and multi-threaded CPU versions of the algorithm with the GPU versions.
3) We demonstrate that GPUs facilitate low-cost real-time resizing of images and videos.

2 Seam Carving

Seam carving [6] transforms the size of images and videos by carving-out pixels that form a path of low-energy. These low-energy connected paths, called *seams*, go from top to bottom or left to right for vertical or horizontal resizing, respectively. The seams are added to or removed from an image[1] in order to increase or reduce its size with minimal observable distortion. Figure 1 shows the steps of horizontally resizing an example image. Since the majority of execution time is spent on the energy function and seam map computations (see section 5), in this paper, we focus on accelerating these two computationally-intensive parts.

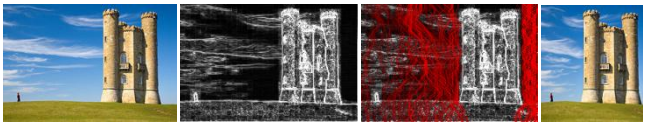


Figure 1[2]: Seam Carving Steps. (a) The original image. (b) The energy of the image using gradient magnitude. (c) The low energy seams with the energy function. (d) Resized image after the seams are removed.

2.1 Energy function

Seam Carving can utilize a variety of energy functions to generate seams [6]. The magnitude of the gradient approach uses equation (1) to compute the energy of each pixel relative

[1] For most of the paper, we only discuss images. However, a video is a set of images or frames displayed at the video rate of 30 frames per seconds. Seam carving is equally applicable to both images and videos.
[2] Image taken from Wikimedia Commons.

to its surrounding pixels by quantifying the amount of change in color from one pixel to the next.

$$e(I) = \left\|\vec{\nabla} I\right\|_1 = \left|\frac{\partial I}{\partial x}\right| + \left|\frac{\partial I}{\partial y}\right| \quad (1)$$

The energy function computation exhibits vast data parallelism. However, attention must be given to memory access patterns, which may have a huge implication on performance. The need for accessing neighboring pixels to compute the energy function strongly influences the way we access memory on the GPU.

2.2 Seam map

After finding the energy of each pixel, the result is used to locate the lowest-energy paths or *seams*. We focus on the implementation of the seams required for horizontal resizing, i.e. the vertical seams. The first row of the seam map is directly obtained from the first row of the energy function. Starting from the second row of the image, we use a dynamic programming approach (2) to compute the seam map value at every pixel.

$$S_{i,j} = \begin{cases} E_{0,j} & \text{if } i = 0 \\ E_{i,j} + \min\left(S_{i-1,j-1}, S_{i-1,j}, S_{i-1,j+1}\right) & \text{otherwise} \end{cases} \quad (2)$$

In equation (2), S is the seam map table, E is the energy function table, and i and j are the rows and columns indices of the tables. This dynamic programming approach produces the optimal seam/s [6]. The values in the final row of the seam map table correspond to the cumulative energy of the seams. The most important point to note about equation (2) is that the computation for each element is entirely dependent on the result of the three elements directly above it, as shown in Figure 2. Therefore, unlike the energy function, the data in the seam map computation is not 100% separable. This makes parallelizing the seam computation much more difficult than the energy function.

3 Hardware resources

All of the implementations presented in this paper were executed on two different heterogeneous computer systems. The First system is a Mac Pro running the Mac OS X 10.6 operating system powered by two 2.8GHz quad-core Intel Xeon E5462s CPUs. The Mac pro has an NVIDIA 8800GT GPU (G80 architecture) with 112 cores and 512MB of GDDR3 memory.

The second system is a newer machine running the Ubuntu Linux 10.04 operating system, powered by a single 3.4 GHz quad-core Intel Core i7 2600k CPU. The GPU on the Linux machine is the NVIDIA GTX580 Featuring the Fermi architecture with 16 SMs (32 cores per SM) for a total of 512 SPs. The GTX580 has 1.5GB of GDDR5 memory, 768KB L2 cache, and 64KB configurable L1 cache/shared memory per SM. The Intel Core i7 supports simultaneous multithreading (SMT) while the Intel Xeon does not support SMT [8, 9].

The parallelization tool for the CPU implementation is POSIX threads (pthread). Both of the CPU implementations were optimized and compiled with *gcc* -O2 optimization level. Finally, the heterogeneous implementations were compiled with the NVIDIA *nvcc* compiler, which uses *gcc* and the -O2 flag to compile the CPU code

4 Implementation

4.1 CPU implementation

4.1.1 Energy function

The single threaded CPU implementation of the energy function utilizes a set of nested *for-loops* to compute each of the partial derivatives (Algorithm 1, lines 1-7), and utilizes the result to compute the magnitude of the gradient (Algorithm 1, lines 8-9). The one-half is factor out of the derivatives and applied after the sum in order to save one multiplication operation per pixel. Instead of directly storing a 2D image in a 2D array, we store the 2D image in a 1D array and map the 1D to a 2D array. This minimizes the allocation time, and the data is stored in a more suitable manner to take advantage of spatial locality. Note that the derivative of each pixel is computed once, the purpose of the nester loops is to ease the computation, but each derivative has O(n) time complexity.

Given that there are no data dependencies in the computation of the energy function, we are able to divide the computation into as many threads as the operating system supports. However, the performance is dictated by the hardware and the CPUs' ability to execute threads simultaneously. We divide the input image into tiles consisting of consecutive rows. The height of each tile is computed based on the number of threads and the height of the image. Figure 3 illustrates the decomposition of the input image for an execution configuration of eight threads. The Algorithm for the multithreaded version is very similar to Algorithm1 with the exception that each thread only loops through its corresponding portion of the image.

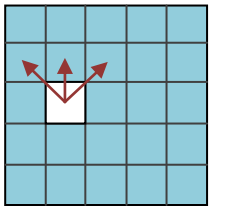


Figure 2. Seam map example.

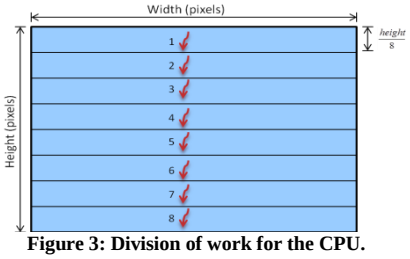


Figure 3: Division of work for the CPU.

Alg. 1 Single threaded Energy function

```
1: Set all elements of Ix and Iy to 0
2:    for i ← 0 to height – 1
3:       for j ← 1 to width – 2
4:          Ix(i, j) ← I(i, j+1) – I(i, j-1)
5:    for i ← 1 to height – 2
6:       for j ← 0 to width – 1
7:          Iy(i, j) ← I(i+1, j) – I(i-1, j)
8:    for all pixel in the image
9:       energy = 0.5 * (|Ix| + |Iy|)
```

Alg. 2 multithreaded Seam map

```
1: for i ← 1 to height – 1
2:    for j ← t_start to t_end
3:       if (j-1 < t_start)
4:          wait for S(i,j-1) to be unlock
5:       if (j+1 > t_end)
6:          wait for S(i,j+1) to be unlock
7:       S(i,j) ← Min(S(i-1,j-1),
            S(i-1,j), S(i-1,j+1))+energy(i,j)
8:       Unlock S(i,j) //initially locked
```

4.1.2 Seam map

Unlike the energy function, the seam map computation uses a dynamic programming approach that is not parallelization-friendly (see section 2.2). Therefore, we have to perform a row-by-row computation of the seam map, which serializes the execution of rows. We achieve parallelism by dividing each row into fixed-width tiles and computing these tiles in parallel. We synchronize all threads after the execution of each row. This method does not yield any significant benefit over the single-threaded version; in fact, with more than two threads, the program spends more time synchronizing than performing the computations.

In an attempt to optimize the seam map implementation, we used locks to create local barriers in place of global barriers. Instead of stalling threads until every thread finishes its part, each thread is only concerned with the execution of its neighboring threads. We therefore use an array of locks to allow each thread to manage the availability of the seam map results of its boundary element on every row. This approach is illustrated in Algorithm 2 where *t_start and t_end* are computed by dividing the width of the image by the number of threads and assigning each thread their respective areas.

4.2 Energy function on the GPU

4.2.1 Naive implementation

The first GPU method presented in this paper is the *naive-non-aligned* implementation. In this implementation, the image was partitioned into 16x16 tiles containing 256 pixels as illustrated in Figure 4a. In addition to loading the corresponding data into shared memory, the kernel also needs to load the pixels immediately adjacent to the tile. These outer pixels are known as the tile apron, shown in white in Figure 4b. Each tile utilizes one 2D block of 324 threads (18x18), thus assigning one thread per pixel load.

All CUDA threads in the Naive kernel execute Algorithm 3. First, each thread calculates the necessary pixel indices to copy a pixel from the global to the shared memory (lines 1-4). Line 6 caches the pixels in shared memory and line 8 ensure that all data transfer completes before performing the computation. Finally, we use 256 out of 324 threads to compute the gradient and store result (lines 10-12).

At first, we used a three-byte data structure to store the RGB components of each pixel. A three-byte data structure causes unaligned memory accesses, which reduces performance. We solved this problem by aligning pixels to word length (4 bytes). This implementation waste 25% of the total memory. However, if the memory size is sufficient, this is an excellent tradeoff. In addition, there are times when we are interested in preserving the alpha component of all pixels; this implementation guarantees that all pixels retain the original alpha value. Another optimization technique is to allocate memory on the device using the *cudaMallocPitch* function [7]. Using the two-dimensional allocation and copy functions (*cudaMemcpy2D*), we guarantee that each row of the image starts on a 64- or 128-bytes boundary in global memory depending on the device architecture.

Alg. 3 Naive implementation of the energy function in CUDA

```
1:   col ← block_x * tile_width + x – 1     // x : thread_x
2:   row ← block_y * tile_height + y –1     // y : thread_y
3:   k ← row * image_width + col
4:   i ← y * bw + x                         // bw : block_width
5:   Ix ← 0,  Iy ← 0
6:   if (k is an index within the image)  SMEM(i) ← image(k)
7:   else  SMEM(i) ← 0
8:   Synchronize_threads
9:   if (x > 0 and x ≤ tile_width and y > 0 and y ≤ tile_height)
10:      Ix ← SMEM(i+1) – SMEM(i–1)
11:      Iy ← SMEM(i+bw) – SMEM(i–bw)
12:      ENERGY(k) ← 0.5 * (|Ix| + |Iy|)
```

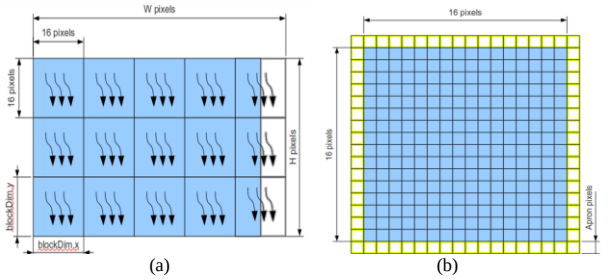


Figure 4: (a) Division of work between threads in the naive GPU implementations; the white area represents idle threads. (b) Partition of block; the tile apron is shown in white and the writable pixels in blue.

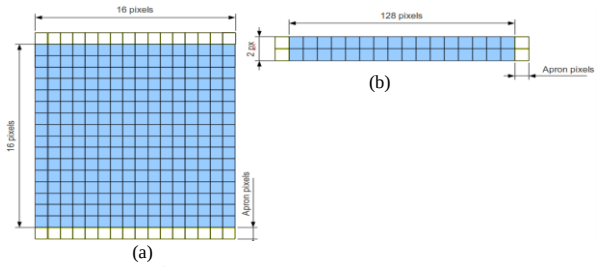


Figure 5: Partition of the image. (a) Vertical tiles. (b) Horizontal tiles. White represents the apron pixels and blue/dark the workable pixels

4.2.2 Split-aligned implementation

To achieve nearly full coalesced memory access, we decided to separate the energy function calculation into two separate kernels, a horizontal and a vertical gradient kernel, and combine the results of the two. This allows us to reorganize the thread grid to suit the kind of memory accesses expected for each direction of the derivatives. Both the apron pixels and workable pixels in the vertical direction are always aligned to 16 pixels as shown in Figure 5a, allowing coalesced accesses of 64 bytes.

For the horizontal calculation, it is not possible with this approach to avoid uncoalesced memory accesses because the apron pixels lie outside the alignment boundary. However, by increasing the tile width to 128 pixels (Figure 5b), some of the wasted bandwidth due to uncoalesced memory accesses is hidden. This improves the memory efficiency allowing only two uncoalesced loads per every eight coalesced loads, and thus increasing bandwidth usage. Algorithm 4 and 5 show the implementation of the *split-aligned* method to compute the energy function.

Alg. 4 Vertical implementation of the gradient in CUDA

```
1:  col ← block_x * bw + x   // x/y : thread_x/y,  bw : block_width
2:  row ← block_y * tile_height + y –1
3:  k ← row * image_pitch + col
4:  i ← y * bw + x,  Ix ← 0,  Iy ← 0
5:  if (k is an index within the image )  SMEM(i) ← image(k)
6:  else  SMEM(i) ← 0
7:  Synchronize_threads
8:  if ( y < tile_height )
9:      row ← row + 1,  i ← i + bw
10:     k ← row * energy_pitch + col
11:     if (row < image_height)
12:         Iy ← SMEM(i + bw) – SMEM(i – bw)
13:         ENERGY(k) ← |Iy|
```

Alg. 5 Split-aligned implementation of energy in CUDA

```
1:  col ← block_x * block_width + thread_x
2:  row ← block_y * block_height + thread_y
3:  k ← row * image_pitch + col
4:  x ← thread_x + 1,  y ← thread_y,  Ix ← 0
5:  i ← y * HORIZ_WIDTH + x
6:  if (k is an index within the image)  SMEM(i) ← image(k)
7:  if (thread_x == 0 and row < image_ height)
8:      pbase ← col * image_pitch
9:      SMEM(y * HORIZ_WIDTH) ← image(pbase + block_width * block_x – 1 )
10:     SMEM(y * 2 * HORIZ_WIDTH – 1) ← image(pbase + (block_width+1) * block_x)
11: Synchronize_threads
12: if (col < image_width)
13:     k ← row * energy_pitch + col
14:     Ix ← SMEM(y * HORIZ_WIDTH + x + 1) – SMEM(y * HORIZ_WIDTH + x – 1)
15:     ENERGY(k)← 0.5 * (|Ix| + ENERGY(k))
```

4.2.3 Locality-aware implementation

The *split-aligned* method has the potential to reduce the number of uncoalesced memory accesses by a significant amount, but it does not make the best use of locality. Therefore, when porting the implementation to the newer heterogeneous system, we decided to revise the *split-aligned* method in order to take advantage of locality and the new capabilities provided by the Fermi architecture. The GTX580 offers approximately 4.5X more SPs than the 8800GT. It also supports memory accesses of up to 128-bytes on a single coalesce load.

Apart from the two outermost pixels that surround the entire image, all pixels are utilized four times in the energy function computation; twice for each of derivatives. Even when these pixels are cached in shared memory, which saves one global load per derivative, the *split-aligned* method requires that each pixel be loaded twice. Therefore, we decided to go back to implementing the energy computation using a single kernel to compute both partial derivatives.

The *locality-aware* method breaks the image into 2D blocks of 512 threads. Each tile contains 64 columns and 8 rows. With a 64x8-block configuration, two warps are assigned per row. All 32 threads in a warp are able to cache their corresponding pixel on a single coalesce load of 128-bytes for 16 fully coalesce loads. Loading the top and bottom aprons also results in fully coalesce loads. The uncoalesce loads are introduced by the left and right aprons of the tile. Increasing the number of rows in a tile improves locality, but increases the number of uncoalesce loads. Increasing the block width minimizes the number of uncoalesce loads, but reduces locality. After careful analysis and performance tests, we found that 8 rows and 64 columns generate the best performance. Algorithm 6 gives an in-depth description of the *locality-aware* implementation of the energy function.

Alg. 6 Locality-aware implementation of the energy in CUDA

```
1:  col ← block_x * block_width + thread_x      // We ensure col
2:  row ← block_y * block_height + thread_y     // and row are within
3:  k ← row * image_pitch + col                 // the image width
4:  j ← thread_x + 1,  i ← thread_y + 1         // and height
5:  SMEM(i, j) ← image(k)
6:  if (thread_x == 0 and col ≠ 0)  SMEM(i, 0) ← image(k – 1)
7:  if (thread_x == block_width–1 and col ≠ image_width–1)
8:      SMEM(i, block_width + 1) ← image(k +1)
9:  if thread_y == 0 and row ≠ 0
10:     SMEM(0, j) ← image(k – image_pitch)
11: if thread_y == block_height–1 and row ≠ image_height–1
12:     SMEM(block_height + 1, j) ← image(k + image_pitch)
13: Synchronize_threads
14: Ix ← 0,  Iy ← 0,  k ← row * energy_pitch + col
15: if pixel are not on the edge of the image
16:     Ix ← SMEM(i, j + 1) – SMEM(i, j –1)
17:     Iy ← SMEM(i + 1, j) – SMEM(i – 1, j)
18: else if pixel is in the first or last rows
19:     Ix ← SMEM(i, j + 1) – SMEM(i, j – 1)
20: else if pixel is on the first or last columns
21:     Iy ← SMEM(i + 1, j) – SMEM(i – 1, j)
22: ENERGY(k)← 0.5 * (|Ix| + |Iy|)
```

4.3 Seam map on the GPU

The GPU implementation of the seam map computation is very similar to the multithreaded CPU implementation, which is described in Section 4.1.2. Each row of the image is broken into horizontal tiles, whose width is carefully selected in order to maximize the occupancy of the GPU. Given that, blocks are not scheduled deterministically and that there is no synchronization among threads on different blocks, we must resort to calling the kernel once per row and synchronize in between calls. For this implementation, wider images should perform much better than narrow images. Similar to the multithreaded CPU implementation, a significant amount of the data is not separable. This limits the amount of parallel execution per kernel launch.

4.4 Page-Locked Memory

The CUDA runtime environment has functionalities to allocate and use *page-locked* memory in place of regular *pageable* host memory [7]. We included this feature in our heterogeneous implementation to optimize the memory transfer. In the performance evaluation, we demonstrate that *page-locked* memory does not affect the performance or results of individual kernels, but it improves the execution time of the memory transfer between the host and device.

5 Performance evaluation

The overall time that it takes to remove a single seam of an image depends highly on the size of the image. The energy function takes the largest fraction of the total execution time, followed by the seam computation. Hence, in this paper, we focus on improving the energy function and the seam map computations. However, we also compare and discuss the total execution times.

5.1 CPU evaluation and results

5.1.1 Energy function

Figure 6 illustrates the performance gained by multithreading the energy function computation and executing the implementation on the Intel Core i7 (4-cores each with SMT) and Xeon CPUs (8-cores, no SMT). The execution of the energy function single-threaded implementation takes 31.5 ms to complete on the Intel Xeon CPU. This is the base system in Figure 6. Results show that the newer Intel Core i7 CPU outperforms the Intel Xeon processor for all of the thread configurations. The best CPU performance for the energy function computation of a 1200x900 image is with the Intel Core i7 and 16 threads. Overall, the energy function computation scales well on multi-core CPUs. With eight cores, the Intel Xeon achieves 7x performance improvement. With four cores and eight hardware threads, the Intel Core i7 achieves 7x speedup over its single-threaded execution. In addition, the Intel Core i7 achieves a 10x performance improvement over the Intel Xeon single-threaded execution. Finally, as the number of threads launched increase beyond the number of hardware threads in the system, the performance gain becomes smaller due to the thread switching overheads. The only exception is the 16-thread execution of the Intel Core i7, which needs further research and it is left as future work.

5.1.2 Seam map

In section 4.1.2, we discussed the implementation of the seam map on the CPU and the dependability among rows of pixels. We emphasized how dependability due to the dynamic programming approach serialized the execution of rows. However, the results exposed another problem that significantly affects the parallelization of the seam map computation. Figure 7 illustrates the performance results of the seam map. The Figure shows that barriers impose a substantial overhead, resulting in a zero gain in performance. In the multithreaded implementation, the performance is worse than that of the single-threaded implementation.

As mentioned in section 4.1.2, a more efficient approach is to synchronize locally instead of at the global level. This implementation performs better because locks inflict less overhead. However, we are only able to achieve 26% and 60% improvement with 2 threads on the Intel Xeon and Intel Core i7, respectively. This speedup is minor in comparison to the speedups achieved for the energy function. Beyond two threads, we see a large drop in performance even though both systems have eight hardware threads.

5.2 GPU performance evaluation

5.2.1 Naive-non-aligned energy function

On the 8800GT, the *naive-non-aligned* method achieves 5.7x and 8x performance improvement over the single-threaded CPU implementation executing on the Intel Xeon and Core i7, respectively, as shown in Figure 8. This performance improvement is similar to that of the multithreaded CPU implementations. However, this implementation does not take advantage of the GPU's wide memory bus. Its memory access patterns are not coalesced due to the data not being aligned. Since the *naive-non-aligned* method only utilizes three bytes per pixel, a warp will only load 96 bytes and a half of a warp will load 48 bytes. Such memory access pattern is not aligned. The *naive-non-aligned* method was initially designed with the G80 architecture in mind. However, with minimum modifications, this implementation yields 89.7x and 62x performance improvement on the Fermi GTX580, over the single-threaded implementation running on Intel Xeon and Intel Core i7, respectively (see Figure 9).

5.2.2 Naive-aligned energy function

The changes to transform the naive method from a non-aligned to an aligned implementation (see Section 4.2.1) improve the performance relative to the single-threaded version from 8x to 18x and 5.7x to 12.4x on their respective systems as shown in Figure 8. Utilizing the CUDA profiler, we were able to determine the remaining source of our performance problems, uncoalesced accesses. The first naive version incurred over 500,000 uncoalesced loads and 300,000 uncoalesced stores for a 1200x900 image (≈ 1 megapixel); the improved aligned version incurred only 100,000 uncoalesced loads and 50,000 uncoalesced stores. This is still significantly more than one would expect, as an image with

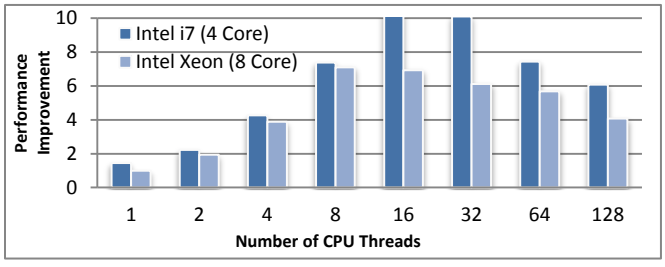


Figure 6: Improvement of the energy function over the single-threaded executing of the Intel Xeon for a 1200x900 image.

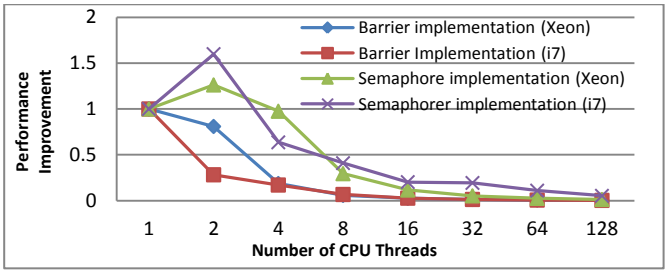


Figure 7: Performance of multi-threaded implementations of Seam map

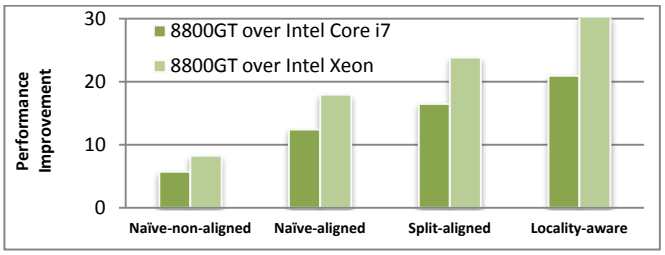


Figure 8: speedup of the energy function over the single threaded CPU

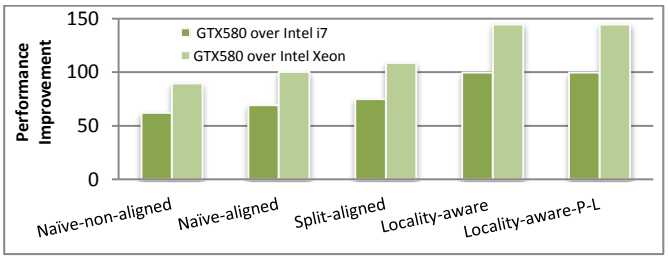


Figure 9: Improvement of the energy function on the GTX580 (Fermi) over the single threaded CPU implementations

this amount of pixels should only need 16,875 loads assuming the GPU can bring in 64 bytes per coalesced loads. The *naive-aligned* method was also designed for the 8800GT. When executed on the GTX580, this implementation shows a performance improvement of 100x and 69x over the respective Intel Xeon and Core i7 single-thread CPU implementations (see Figure 9).

5.2.3 Split-aligned energy function

The *split-aligned* method described in Section 4.2.2 achieves an average of 850 megapixels per second throughput, a 24x and a 16.5x improvement over the single-threaded CPU version on the Intel Xeon and Core i7, respectively, as shown in Figure 8. As expected, the CUDA profiler reveals that for a 1200x900 image, approximately, only 31,000 loads and 15,000 stores were needed (each pixel must be loaded from global memory twice; once for each directional kernel), reducing the total memory access latency by an order of magnitude. On the GTX580, the *split-aligned* achieves 108.6x improvement over the Intel Xeon CPU and 75x over the Intel Core i7 as shown in Figure 9.

5.2.4 Merging the split-aligned method

The l*ocality-aware* method described in section 4.2.3 achieves the highest performance improvement on both GPUs for the energy function computation. By merging the computation of the two derivatives in a way that the number of coalesce loads remains close, and by further taking advantage of locality of accesses, we manage to improve the performance of the energy function by 144.5x and 100x over the single-threaded CPU version on the Intel Xeon and Core i7, respectively, as shown in Figure 9. In addition, when executed on the 8800GT, this method shows a performance improvement of 30x and 21x over the Xeon and Core i7 single-threaded version (Figure 8).

5.2.5 Seam map

The seam map GPU implementation exhibits approximately 4x performance improvement over the single-threaded CPU implementation on the Intel Xeon and no improvement over the Intel Core i7 single-threaded implementation (figure not shown). This performance gain is relatively small in comparison to the energy function speedup. The performance is heavily impacted by the profound dependability among rows in the image. This limits the amount of parallel computation by serializing the execution of rows. Another significant performance impact is the lacking of optimal methods for synchronizing threads among different blocks. Launching the kernel 899 times for a 1200x900 image imposes a significant overhead. Approximately 57% of the seam map execution time is due to kernel launch overhead. Minimizing the launch overhead could potentially improve the performance by a factor of two.

5.3 Evaluation of total execution time of the resizing operation on the GTX580

As previously stated, the energy function and seam map computations account for the largest fraction of the execution time of seam carving. Therefore, by improving these two parts, one would normally achieve a high overall performance improvement. However, there is a penalty when performing computation on the GPU device. The data must be copy from the host memory to the device memory. Once the computation is performed, we must copy the results back to the host memory; if we care to use the results on the CPU side. Both of these operations introduce additional overhead. For extensive GPU computation, the overhead is easily hidden. However, this is not the case for seam carving given that the computations are in the order of micro and milliseconds.

In order to use this GPU implementation of the seam carving in a real word application, we need to utilize the operation described above. Therefore, we need to incorporate the total time that it takes to copy the image from the host to the device, compute both the energy function and the seam map, and copy the result back to the host memory. Figure 10 illustrates the total time that the entire operation takes on the Intel Core i7 and on the GTX580, respectively. This heterogeneous system is selected because it performs the best for both the CPU and the GPU. Figure 11 shows the performance improvement for the entire operation.

Figures 10 and 11 illustrate that the GPU methods perform better than the CPU methods, especially when the size of the image increases. Overall, Figure 11 shows that the total execution time of the best resizing implementation on the GTX580 is about 6x faster than the best single-threaded CPU implementation and over 2x faster than the best multithreaded CPU implementation. The best execution time on entire operation is achieved with the *locality-aware* method using *page-locked* memory. The reason is that the CUDA run-time environment can optimize the host to device and device to host memory copy if the CPU memory is allocated as *non-pageable* memory (see [7]). We therefore modified our fastest implementation, *locality-aware*, to take advantage of *page-locked* memory, which yields the best overall performance as shown in Figures 10 and 11.

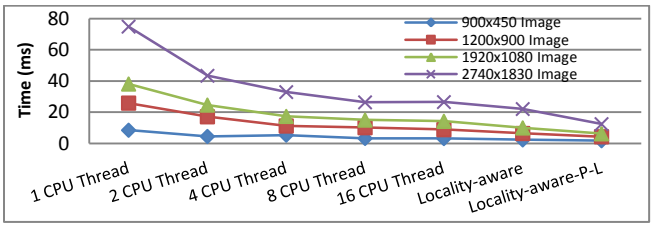


Figure 10: Total time to copy to and from the device, compute the energy function and seam map on the GTX580 and Core i7

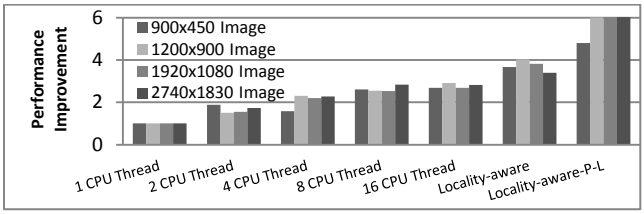


Figure 11: Total time to copy to and from the device, compute the energy function and seam map on the GTX580 and Core i7

6 Related work

Resizing images and videos have been studied extensively in the literature. One of the most popular approaches is to perform cropping [1-5], which involves finding the best rectangular sub-window in the image. However, cropping may lose an unacceptable amount of visual information when important structures lie at all edges of an image. Scaling methods, with or without interpolation, tend to produce distorted images, especially when an image is scaled in one direction.

Avidan and Shamir [6] recently provided a new approach to image and video resizing, called *seam carving*. Seam carving is an algorithm for content-aware resizing of images and videos with little to no perceptible distortion. Seam carving is a computationally-intensive method, which makes it difficult to perform on large images or videos in real-time.

To the best of our knowledge, this paper is the first to implement a real-time content-aware resizing method on GPUs. Our implementation works very well on computing the energy function (over 100x and 144x is possible), but the other computationally-intensive part, *seam map*, is implemented using dynamic programming which limits the amount of data parallelism that can be exploited (only 4x speedup over the Intel Xeon and no improvement over the i7). A recent work [10] implemented a faster way to compute the seam map by finding the optimal matches within a weighted bipartite graph composed of the pixels in adjacent rows or columns. In future work, we will adapt this method, which we believe will improve our results greatly for the seam map computation.

7 Conclusion and future work

Seam carving is a powerful method for resizing images and videos. This content-aware resizing method has been shown to effectively resize images and videos with little to no perceptible distortion. However, the seam carving algorithm is computationally-intensive and for high-resolution images and videos, it may become impossible to perform the resizing in real-time by using the CPUs in a desktop-scale computer.

In this paper, we exploit the highly parallel computational capabilities of CUDA-capable GPUs in a heterogeneous computer system for accelerating the resizing of videos and images through seam carving. Out of the four different GPU methods that we implemented, our results show that the best is the *locality-aware* method using *page-locked* memory, which achieved a performance improvement of 100x over the best single-threaded execution time and 14x over the best CPU multithreaded version of the energy function executing on the Intel Core i7. Overall, our results show that the GPU-based implementation has a significant impact on the performance of seam carving and has the potential to resize videos and large images in real-time.

In the future, we are planning to vectorize the CPU implementation to take advantage of the SIMD instructions on the Intel CPUs. Another important part of our future work is to find a better approach to parallelize the seam map computation.

8 Acknowledgement

We would like to thank the anonymous reviewers for their efforts. In addition, we would like to thank Harry Bock for his contributions. This work was supported in part by US National Science Foundation grant CCF-1117467.

9 References

[1] Itti L, Koch C, Niebur E, "A model of saliency-based visual attention for rapid scene analysis," IEEE Trans Patt Anal Mach Intell, 1998, Vol. 20, No. 11, pp. 1254–1259

[2] Sue B, Ling H, Bederson B, et al. "Automatic thumbnail cropping and its effectiveness," In: Proc. of User Interface Software and Tech., 2003, pp. 95–104

[3] Chen L, Xie X, Fan X, et al. "A visual attention model for adapting images on small displays," Multimedia Syst, 2003, Vol. 9 No. 4, pp. 353–364

[4] Ciocca G, Cusano C, Gasparini F, et al. "Self-adaptive image cropping for small displays," IEEE Trans Consumer Electr, 2007, Vol. 53 No. 4 pp.1622–1627

[5] Santella A, Agrawala M, DeCarlo D, et al. "Gaze-based interaction for semi-automatic photo cropping," Proc. of Human Factors in Comp. Sys, 2006, pp. 771–780

[6] S. Avidan, A. Shamir, "Seam Carving for Content-Aware Image Resizing," In SIGGRAPH '07 ACM SIGGRAPH, 2007.

[7] NVIDIA, "NVIDIA CUDA C Programming Guide 4.2," (2012). [online]. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[8] Intel, "2nd Generation Intel Core Processor Family Desktop," (2011). [online]. Available: http://www.intel.com/content/www/us/en/processors/core/2nd-gen-core-desktop-vol-1-datasheet.html

[9] Intel, "Quad-Core Intel Xeon Processor 5400 Series, Datasheet" (2008). [Online]. Available: http://www.intel.com/assets/PDF/datasheet/318589.pdf

[10] H. Hua, F. TianNan, R. Paul L, Q. Chun, "Real-time Content-aware Image Resizing," Science in Chine Series F: Information Sciences, 2009, Sci. in China Press.

A Novel Quantum-dot Cellular Automata Switch for Field Programmable Gate Arrays

Hemant Balijepalli[1] **and Mohammed Niamat**[1]
[1]Dept. of Electrical Engineering and Computer Science, University Of Toledo, Toledo, Ohio, USA

Abstract - *In this paper, we propose a novel Quantum-dot Cellular Automata (QCA) based Programmable Switch Matrix which can be used as a routing element for future nano Field Programmable Gate Arrays (FPGAs). QCA technology is an emerging technology based on encoding binary information in charge configuration within quantum dot cells. In FPGAs, the Programmable Switch Matrix (PSM) is the interconnection circuit which switches the signals at the intersection of the horizontal and vertical routing channels. The main goal behind the design of QCA based PSM is to design a routing element for transistor-less nano FPGAs. The proposed QCA based Programmable Switch Matrix (QPSM) has an advantage over the area and the number of cells compared to earlier designs. The proposed QPSM is implemented and simulated using the QCA Designer tool.*

Keywords: Quantum-dot Cellular Automata, Field Programmable Gate Arrays, Programmable Switch Matrix.

1 Introduction

Moore's law states that the number of transistors on a unit area doubles every 24 months [1]. Over the decades, the exponential scaling of the feature size and increase in the processing speed has been provided by CMOS technology for implementing the VLSI systems [2]. The CMOS technology due to the reduction of the feature size is facing consequences like power dissipation, diminishing returns in switching performance, diffusion barriers, gate depletion, stray capacitances and electro-migration [3, 4]. Various technologies have been investigated in the past to find a potential replacement for the CMOS technology in the future. Some of these technologies are Single Electron Transistors (SETs), Resonant Tunneling Diodes (RTDs), Carbon Nano Tubes (CNTs), and Quantum-dot Cellular Automata (QCA) [5, 6]. Proposed by Lent and others in 1993, QCA technology uses arrays of coupled quantum dots to implement the boolean logic functions. Each QCA cell comprises of four electron wells and two free electrons. The electrons can tunnel between the dots through electron tunneling [7]. Due to Coulombic repulsion, the electrons repel each other and align themselves in the opposite corners of the cell. The electron configuration on the diagonals lead to two different state polarizations which are labeled as logic '0' and logic '1', as shown in Figure 1. The cell with polarization -1 represents the binary state 0 and the cell with polarization +1 represents binary state 1. The major advantages of QCA are the small circuit size, higher clock frequency, and lower power consumption.

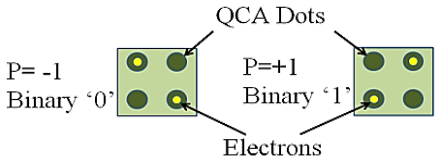


Figure 1. Logic States of QCA

By arranging the QCA cells in different ways, QCA devices such as QCA wires, QCA inverters and Majority voters can be designed (Figure 2). The QCA Majority voter (MV) has logic function MV (A, B, C) = AB + BC + AC. The QCA based AND gate and the OR gate can be designed by setting one of the inputs of MV to 0 (P=-1) and 1 (P=+1), respectively.

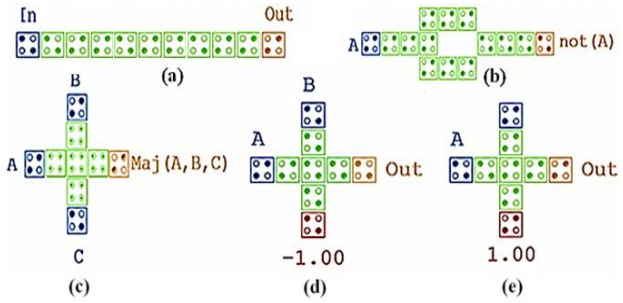


Figure 2. Basic QCA Devices a) Binary QCA Wire b) QCA Inverter c) Majority Voter Gate d) AND Gate e) OR Gate

In QCA, individual wires and gates are clocked. The clocking amplifies the signal and provides the direction of flow to the signal. The clock wires are run underneath the layer consisting of the QCA cells. The clock in QCA is divided into four phases with each phase incurring a 90^{o} phase delay. The four clocking zones are named Switch, Hold, Release and Relax [8] as shown in Figure 3.

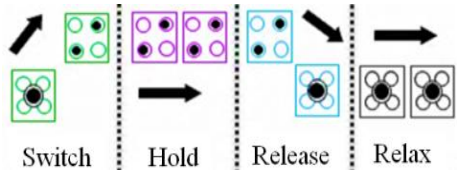


Figure 3. QCA Clock Phases

The clocking scheme allows the arrays of cells to perform the computations based on the position of inter dot

potential barriers and the result is given as the input to the successor array. If the potential barrier is held high, the cell gets polarized and if the potential barrier is lowered, the cell becomes unpolarized. The cell switches to polarized state in the switch phase, remains polarized in hold phase, becomes unpolarized in release phase and finally stays unpolarized in the relax phase until it gets polarized in the next switch state. In this manner, the information is transferred through QCA cells by maintaining the ground state polarizations.

The QCA based routing elements of an FPGA have been proposed in the past. Niemer and others have proposed the design for FPGA logic block and interconnects in QCA [9]. The work in [9] explains that there is no possibility to create equivalent pass transistors using the QCA devices. Thereby they have defined unclocked zones where the cells can remain unpolarized. This is possible by unclocking the cells by fixing their barrier potential to a certain value or by keeping the cells in release phase of the clock. Niemer designed routing elements of various sizes using the unclocking concept. Based on a similar idea, Andrej and others have proposed a basic universal crossing element in QCA [10] which can be used as a switch matrix in FPGAs. In this paper, we propose a novel QCA based Programmable Switch Matrix of an FPGA by keeping the cells in a particular region in the release phase of the clock. Initially, a (1x1) QCA Programmable Switch Matrix (QPSM) is designed based on which switch matrices of any size can be designed. Size of the switch matrix is defined by the number of inputs and outputs of the switch matrix. The concept of coplanar crossover of QCA wiresis also used. The designs are simulated using the QCA Designer tool [11].

The rest of the paper is organized as follows: Section II details the design of the different QCA Programmable Switch Matrices. The simulation results are illustrated in Section III. Section IV includes the conclusion.

2 Design of a QCA Programmable Switch Matrix (QPSM)

The Field Programmable Gate Arrays consists of arrays of logic blocks (cells) placed in an infrastructure of interconnections. The FPGAs can be programmed at the logic cells, at interconnections between the cells, and at inputs and outputs [12]. The major elements of an FPGA are the Configurable Logic Blocks (CLBs), Programmable Switch Matrices (PSMs) and Input and Output blocks (IOBs). The CLB is a basic programmable block implementing digital logic in an FPGA. The CLBs can be configured to perform basic logic functions using the Look up Tables (LUTs). The CLBs are interconnected through Programmable Switch Matrices to form the units which can perform complex functions. The I/O blocks provide programmable I/O connections between the FPGA and the peripherals of the system. It is anticipated that future nano FPGAs will find applications in fields such as telecommunication, medicine, aerospace and industrial control.

In FPGA, the Programmable Switch Matrix is used to switch signals at the intersection of horizontal and vertical routing channels. The switch matrix establishes a communication between different elements of an FPGA. The PSM is used to make the required connections between various logic blocks and input output blocks. Today, many commercially available FPGAs use island-style architecture in which logic blocks are organized in an array separated by horizontal and vertical routing channels (Figure 4). The CLBs are connected to the horizontal and vertical lines using a connection block (C block) and the signals are switched at the intersection of channels by a switch matrix (PSM block) [13] as shown in Figure 4.

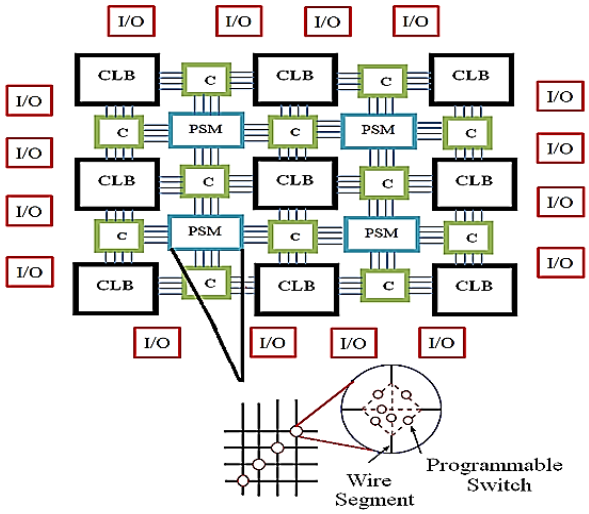


Figure 4. Island style architecture of an FPGA

In the PSM, the crossway consists of six pass transistors that lets the signal flow in either of the four possible directions. In earlier research described in [9,10], QCA based interconnects and routing elements were designed by proposing unclocking select cells or setting the cells in the release phase of the clock. However, the circuits were not simulated. Based on the concept of keeping cells of a certain region in the release phase, we propose a novel design of a QCA Programmable Switch Matrix (QPSM). Using QPSM, the signals can be transferred from one wire to another wire of an FPGA in all the possible directions. As the QCA Designer tool does not support unclocking, we have designed the switch matrices by selectively setting the group of cells in the release phase of the clock.

The (1x1) QCA Programmable Switch Matrix (1-QPSM) is used to switch signals between single horizontal and vertical channels. The PSMs designed in this work have the vertical channel as QCA inverted wire and the horizontal channel as normal QCA wire. When a normal QCA wire crosses an inverted QCA wire, there is no interaction between them. This is possible as the energy between the standard cell and the rotated QCA cell cancels out making the kink energy zero. This crossover of wires is called the coplanar crossover [14]. The two wires are connected by

tapping a normal cell at half way of the rotated cells of QCA inverted wire and connecting it to the normal QCA wire. This is designated as a QCA Connector in this paper. For the signals to flow on the respective individual wires, the connector cells are kept in the release phase of the clock so that they do not affect the cells of either of the wires. For the signals to flow from one wire to the other, correct clocking must be provided to the QCA Connector. In Figure 5, the QCA Connector cells are kept in the release phase so that the inputs In1 and In2 appear at the outputs Out1 and Out2, respectively. Based on the (1x1) QPSM, (n x n) QPSMs can be designed where 'n' is the size of the switch matrix.

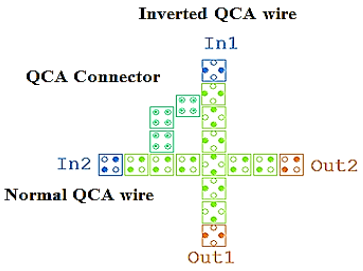


Figure 5. A (1x1) QCA Programmable Switch Matrix (1- QPSM)

In Figure 6, the design of a (4x4) QCA Programmable Switch Matrix (4- QPSM) is shown. The input signals In1-In4 can be routed in vertically up, down and horizontal directions. By providing the respective clock to the QCA Connectors (dotted purple box), the signals travel from the horizontal channels to vertical channels. The signals are then transferred in up and down direction by clocking the appropriate cells in the vertical channel. The other QCA Connectors are kept in the release phase of the clock so that they do not affect the signals flowing on the horizontal channel. Therefore, the signals are transferred from one wire to the other by providing the clock in the direction the signal has to flow and keeping the rest of the cells in the release phase of the clock.

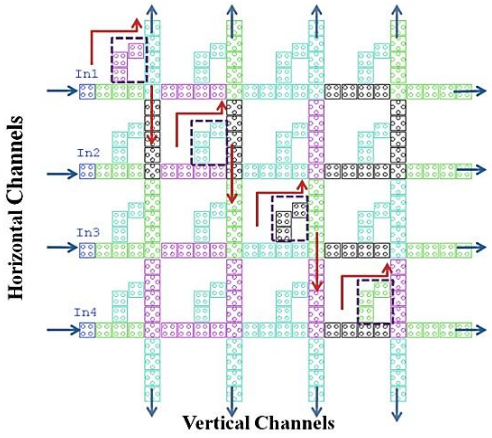


Figure 6. A (4x4) QCA Programmable Switch Matrix (4- QPSM)

In FPGA, the Connection Block connects the horizontal and vertical channels to the CLBs. We have designed the QCA Connection Blocks (QCB) to provide these connections. A four input QCB (4-QCB) is shown in Figure 7. The inputs In1-In4 on the horizontal channels can be routed in three directions (straight, up and down). Select QCA Connectors are clocked to allow the signals to flow in the desired direction, the rest of the Connectors are kept in the release phase. For example, if the signals are supposed to connect to the CLB at the top and not to the CLB at the bottom, the QCA connectors above the horizontal channels are clocked and the QCA connectors below the horizontal channels are kept in the release phase. Similarly, the QCB which connects the vertical channels to the CLB can be designed.

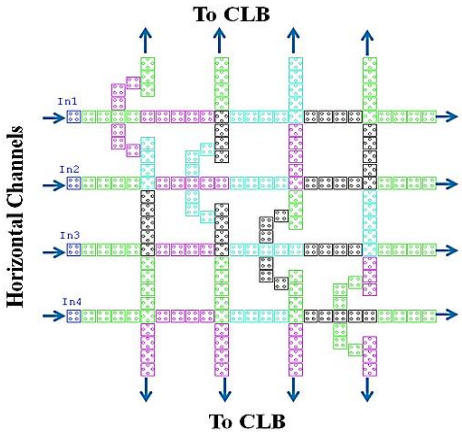


Figure 7. A (4x4) QCA Connection Block (4- QCB)

A layout of an island style FPGA with the proposed QPSMs is shown in Figure 8. In the figure, PSM1-PSM4 represents the 4-QPSMs and C1-C5 represents the 4-QCBs. The inputs are shown by blue arrows and the outputs are shown by orange arrows.

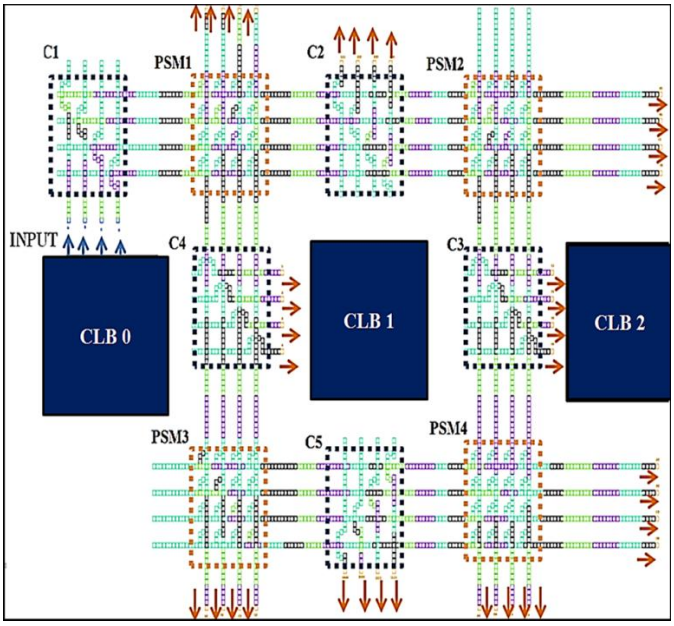


Figure 8. QCA layout showing the QCA Programmable Switch Matrices (QPSM) and QCA Connection Blocks (QCB)

To verify the functionality of the proposed QPSM, signals are transferred in multiple directions in the layout. Table I shows the flow of the signal through each Connection block.

TABLE I. FLOW OF THE SIGNALS IN THE QCA LAYOUT

Block	Flow of the signal	Block	Flow of the signal
C1		PSM1	
C2		PSM2	
C3		PSM3	
C4		PSM4	
C5		-	-

3 Results

The proposed QPSMs are simulated using the QCA Designer tool for functional verification. In Table II, the number of QCA cells and the area occupied by different sizes of QPSMs are listed. It is found that, for a QPSM of size 'n', the number of QCA cells required are given by 4n (3n+1) cells and the area occupied is given by [n (n+1) -1] / 100 (μm^2). The simulation result of a 4- QPSM is shown in Figure 9. For the inputs In1-In4, the QPSM gives the correct outputs Out1-Out4 (indicated by the arrows). Similarly, the input signals can be switched in other directions and the expected outputs obtained.

TABLE II. NUMBER OF CELLS AND AREA OF EACH SWITCH MATRIX

Size of the QPSM (n)	Number of QCA cells = 4 n(3n+1) cells	Area = $\frac{n\,(n+1)-1}{100}$ (μm^2)
1	16	0.01
2	56	0.06
3	120	0.11
4	208	0.19
5	320	0.29
6	456	0.41

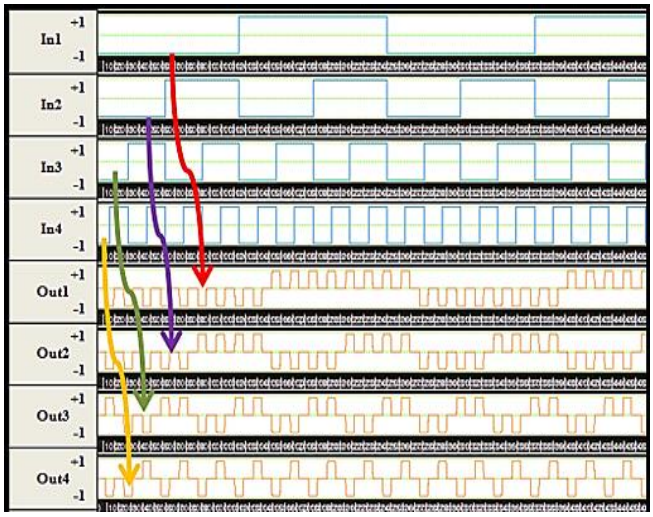


Figure 9. Simulation result of 4-QPSM

4 Conclusion

In this paper, design of a QCA based Programmable Switch Matrix is proposed for nano Field programmable Gate Arrays. The proposed PSMs can be used as potential routing elements in the 'beyond CMOS era'.

5 Acknowledgement

This research was partially supported by NSF grant # 0958298.

6 References

[1] P.K. Bondy, "Moore's law governs the silicon revolution," in *Proceedings of the IEEE*, vol. 88, pp. 78-81.

[2] International Technology Roadmap for Semiconductors (ITRS), "http://www.itrs.net/Links/2011ITRS/Home2011.htm," 2011 Edition.

[3] J. Fortes, "Future challenges in VLSI system design," *IEEE Computer Society Annual Symposium on VLSI*, pp. 5–7, 2003.

[4] D. J. Frank, R.H. Dennard, E. Nowak, P.M. Solomon, Y. Taur, and S.P. Wong, "Device Scaling Limits of Si MOSFETs and Their Application Dependencies," *Proceedings of the IEEE*, vol.89, pp. 259 –288, 2001.

[5] C. S. Lent, "Molecular Electronics: Bypassing the transistor Paradigm," *Science*, vol. 288, pp. 1597- 1599, June 2000.

[6] T. D. Tougaw and C. S. Lent, "Logical devices implemented using quantum cellular automata," *Journal of Applied Physics*, vol.75, no. 3, pp. 1818-1825, Feb. 1994.

[7] G. Pallav, K. J. Niraj, and L. A. Lingappan, "Test Generation Framework for Quantum Dot Cellular Automata Circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no.1, pp. 24-36, Jan. 2007.

[8] M. Askari and M. Taghizadeh, "Logic Circuit Design in Nano-Scale using Quantum-Dot Cellular Automata," *European Journal of Scientific Reserch*, vol. 48, pp. 516-526, 2011.

[9] M. T. Niemer, A. F. Rodrigues, and P. M. Kogge, "A Potentially Implementable FPGA for Quantum Dot Cellular Automata," *1st Workshop on Silicon Computation (NSC – 1)*, Boston, MA, pp. 38-44, Feb. 2002.

[10] J. Andrej, Z. Nikolaj, L. B. Iztok, P. Primoz, and M. Miha, "Quantum Dot Field Programmable Gate Array: enhanced routing," *Conference on Optoelectronic and Microelectronic Materials and Devices*, Perth, Western Australia, pp. 121-124, Dec. 2006.

[11] K. Walus, T. Dysart, G. A. Jullien, and R. A. Budiman, "QCADesigner: A rapid design and simulation tool for quantum dot cellular automata," *IEEE Transaction on Nanotechnology*, vol.3, no.1, pp. 26-31, March 2004.

[12] P. Marchal, "Field-programmable gate arrays," *Communications of the ACM*, vol. 42, no.4, pp. 57-59, Apr. 1999.

[13] R. Jonathan and B. Stephan, "Flexibility of Interconnection structures of Field Programmable Gate Arrays," *IEEE Journal of Solid State Circuits*, vol. 26, no. 3, March 1991.

[14] G. Schulhof, K. Walus, and G. Jullien, "A Simulation of random cell displacements in QCA," *ACM Journal of Emerging Technologies in Computing Systems*, vol. 3, no. 1, Jan. 2007.

The multi-GPU System with ExpEther

Shimpei Nomura[1], Tetsuya Nakahama[1], Junichi Higuchi[2]
Jun Suzuki[2], Takashi Yoshikawa[2] and Hideharu Amano[1]
[1]Graduate School of Science and Technology, Keio University, 3-14-1 Hiyoshi Kouhoku-ku
Yokohama, Kanagawa 223-8522, Japan
[2]System Platforms Research Laboratories, NEC Corporation 1753
Shimonumabe, Nakahara-Ku, Kawasaki, Kanagawa 211-8666, Japan
Email: cell@am.ics.keio.ac.jp

Abstract— *Clusters using multiple GPUs have been already widespread to build a high performance computer economically. However, since the number of plugged GPUs into a CPU is limited, such clusters are consisting of multiple host PCs each of which has a few GPUs. This conventional multi-GPU cluster requires programmers to learn parallel programming skills for controlling communication between nodes as well as GPU programming.*

In order to show the illusion that a large number of GPUs to a single host, a multi-GPU system with ExpEther is proposed. The multi-GPU system allows interconnecting a single host PC and multiple GPUs by ExpEther which extends PCIe interface to Ethernet.

Execution of the application program with two to six GPUs achieved 1.99, 2.96, 3.92, 4.83 and 5.14 times speedup at most, as those with a single GPU. Also, the influence of the bandwidth of the network used in the multi-GPU system is evaluated quantatively.

Keywords: Graphics processing unit, cluster, Parallel Computing, Scalability

1. Introduction

The GPGPU (General Purpose Computing on Graphic Processing Units) has become a major way for high performance computing. Recent GPUs have multiple SIMD (Single Instruction Multiple Datastreams) units each of which provides more than hundreds processors, and support thousands of concurrent threads. GPUs are much superior to general purpose multi-core CPUs from the viewpoint of both performance per cost and performance per power. Also, their growth of performance per year is also much higher than that of CPUs.

Although the programming of GPUs was difficult for common programmers, it has been rapidly improved. For instance, NVIDIA and AMD support c-like GPU programming languages, Compute Unified Device Architecture (CUDA) [1] and ATI Stream SDK [2], respectively. Open Computing Language (OpenCL) [3] has been widely spread as a programming environment for various accelerators including GPUs. Such programming environment lowers the barrier for introducing of GPU in the many fields.

In order to obtain the performance beyond a single GPU, clusters with GPUs are popularly used in the field of high performance computing. Japanese Tsubame 2.0 [4] system, a supercomputer using GPUs is in the Top500 of would supercomputers ranking. Generally, these multi-GPU systems are consisting of a large number of network connected PCs each of which provides a few GPUs plugged into each slot. This structure comes from that the the GPU needs support of CPU to control data transfer and kernel execution.

This conventional multi-GPU system cause two problems. The first is an increase of the latency of the communication between nodes. The communication between GPUs must be done via its connected CPU, and it often stretches the latency and limits the bandwidth. When the latency of the communication is large compared with the computation time of GPUs, the time for communication can bottleneck the system, especially when the number of nodes becomes large. The second is the programming complexity of parallel processing. In order to use such a cluster efficiently, programmers are required to describe hierarchical parallel programming: that is, the coarse grained parallel programming which controls the communication between GPUs in MPIs or other message passing library and the fine grained parallel programming for intra-GPUs in GPU specialized programming language such as CUDA and OpenCL. Although some tools have been developed to reduce the multi-GPU systems programming difficulty[5][6], they require some performance overhead.

We address these problems by using ExpEther, the Ethernet-based virtualization technology. ExpEther extends PCI Express(PCIe), the standard interface used for connecting hosts and GPU devices to Ethernet. It provides a transport function for PCIe packets by encapsulating them within an Ethernet frame and tunneling between the con-

nected modules.

The proposed multi-GPU system with ExpEther, which is called GPU-BOX, has only a single host PC connected with a large number of GPUs by the Ethernet consisting of conventional switches and cables. GPU-BOX provides PCIe ports and power supply for GPUs together with the function of ExpEther. This system releases programmers from the requirement to learn communication programming between nodes, and enables users to select the number of GPUs independent of a host PC's capacity. Moreover, the latency for communication between GPUs is not so stretched, since the data is communicated between GPUs without using CPUs.

The rest of this papers is organized as follows. We introduce some related work about multi-GPU systems in Section 2. The key technology, ExpEther is explained in Section 3, and then a multi-GPU system by using GPU-BOX is proposed in Section 4. In Section 5, our experimental results are shown. Finally, we conclude this study with future work in Section 6.

2. Related Works

There are some equipments to increase the number of GPUs which a single host PC provide. PCI Express Switches provided by some vendors such as IDT[7] increase PCIe slots of the host PCs. They enable host PCs to connect GPUs over the number of slots provided with host PCs. Also, PCI-SIG announced the availability of the PCI Express External cabling 1.0 specification[8]. It focuses on the implementation of cabled PCIe. Based on the specification, there are external expansion units of PCIe, for instance CONTEC provides some of bus expansion units for PCI Express[9]. However, compared with the system connected by network, extending with these equipments has less flexibility and extensibility.

Some tools for reduction of the complexity of parallel programming are also developed. Vegeta[5] and Hybrid OpenCL[6] virtualize the communication between nodes. By using them, programmers can use multi GPUs without describing communication program in message passing library, but they require some performance overhead. FLAT[10] also has the same policy, though the target program is described in CUDA instead of OpenCL which is the target of Vegeta and Hybrid.

Our proposed multi-GPU system has only a single host and is connected by Ethernet. It doesn't require to describe communication program between nodes since a single host is used. The flexibility and extensibility are not degraded compared with conventional systems.

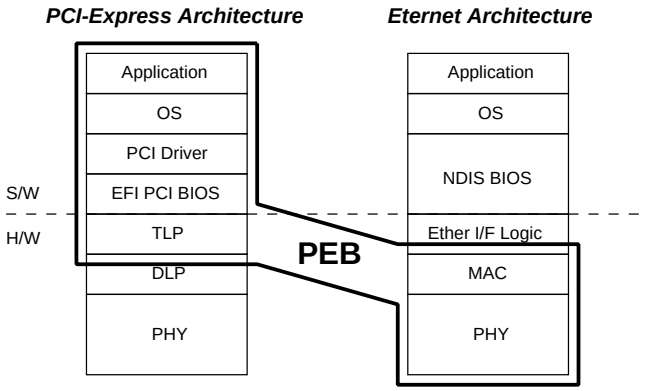


Fig. 1: Overview of PEB

3. ExpEther

ExpEther[11], a key technology of multi-GPU system, is the Ethernet based Virtualization technology developed by NEC[12]. It extends PCIe which is high performance I/O network but limited to small area around the cabinet to much wider area by using Ethernet. Various types of I/O devices on distant location can be connected as if they were inside the cabinet.

In this section, we describe the function of ExpEther, PCI Express-to-Ethernet bridge (PEB) and and Ether-Forwarding-Engine (EFE).

3.1 PEB

As Fig. 1 shows, PEB is the function of ExpEther to bridge the TLP layer in PCIe and the MAC layer in Ethernet. PEB encapsulates a PCIe packet, Transaction Layer Packet (TLP) into Ethernet frame, and decapsulates it for extending PCIe to Ethernet. Moreover, the target Ethernet is virtualized so that the connected devices can be treated as if they were plugged into host PCs without Ethernet. PEB is implemented with a conventional operating system, device driver, PCIe interface, Ethernet Switch and others. That is, the ExpEther technology can be easily introduced into the systems only with PCIe interfaces.

3.2 EFE

EFE is the delay-based protocol which supports repeat and congestion control instead of TPC, the loss-based protocol. The congestion control system in EFE consists of the following steps.

1) It sends a certain number of probe packets, when communication starts.
2) It decides the initial transmission bandwidth based on the acknowledging packets for the probe packets.

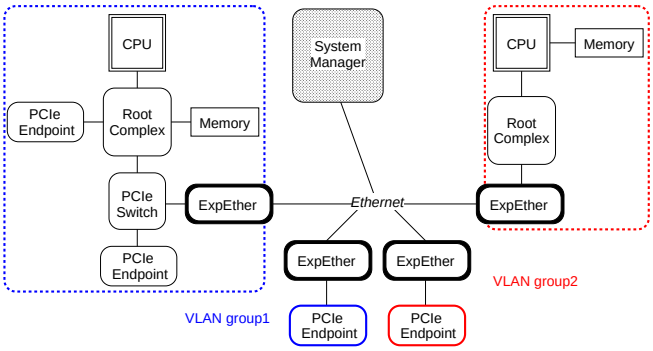


Fig. 2: Example of the System with ExpEther

3) After this, it adjusts the transmission bandwidth for RTT.

EFE employs Go-back-N as a flow-control method. Compared with Selective-Repeat, Go-back-N has the advantage of hardware cost which does not need the reorder buffer. Also, although a number of retransmitted packets are required in the environments of low round-trip propagation delay which ExpEther targets, packet retransmission would not frequently happen.

3.3 System Example using ExpEther

Fig. 2 shows an example of the basic system with ExpEther. On the system of ExpEther, servers and PCIe endpoints are managed and grouped by VLAN ID, which each server or endpoint has in the register of ExpEther bridges. The VLAN ID group consists of a single server and multiple PCIe endpoints. The system manager allocates servers and endpoints the VLAN ID, and can flexibly constitute the groups. On the data transfer, the Ethernet frame has the VLAN ID tag which is referenced to distinguish the group to which the frame belongs.

Though ExpEther doesn't support direct CPUs connection, it can be done by using Remote Direct Memory Access (RDMA). By using the RDMA, CPUs can communicate each other by accessing memory of another CPU directly. It provides low latency and high throughput communications, and suppresses the CPU workload.

4. Design and Implementation

In this section, we propose a multi-GPU system with ExpEther, and show an implementation using GPU-BOX.

4.1 System Design

Fig. 3 shows an overview of the multi-GPU system with ExpEther. It allows to interconnect a single host PC and multiple GPU devices with a common Ethernet through the

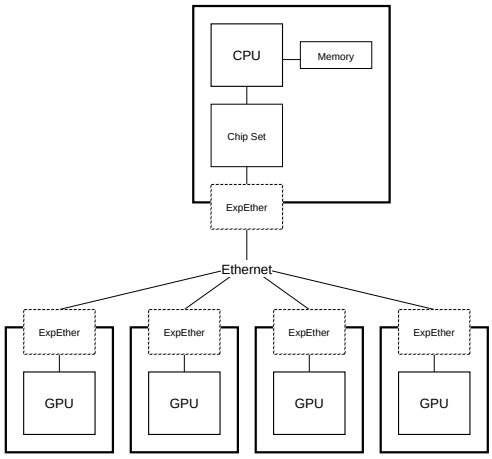


Fig. 3: Multi-GPU System with GPU-BOX

PEB. The network consists of the common Ethernet equipments such as switches and cables. Programmers can treat the multi-GPU system as if all of GPU devices were directly ported into a single host PC, since the PEB encapsulates communication for the control via the network. On the multi-GPU system with only a single host, programmers can make the use of computing power of multiple GPUs without learning programming skill of communication between nodes such as MPI.

The following is a list of the benefits and advantages that the multi-GPU system with ExpEther provides.

- Performance: Stretching the latency of communication between GPUs is suppressed by using ExpEther.
- Programmability: Even programmers who can't describe communication between GPUs can use the system with multiple GPUs.
- Portability: The existing GPU program can run with only a small change about the number of devices in GPU programming language for instance CUDA.
- Flexibility: When the number of GPUs is changed, the application on multiple GPUs can run with only program modification of device size and extending network with Ethernet switches.
- Compatibility: GPUs are accessible to a conventional operating system, device driver, PCIe interface, and Ethernet switch. Thus, they can be used without special modification.
- Future Prospect: The proposed multi-GPU system can receive the benefit of Ethernet technology improvement which will constantly continue in future.

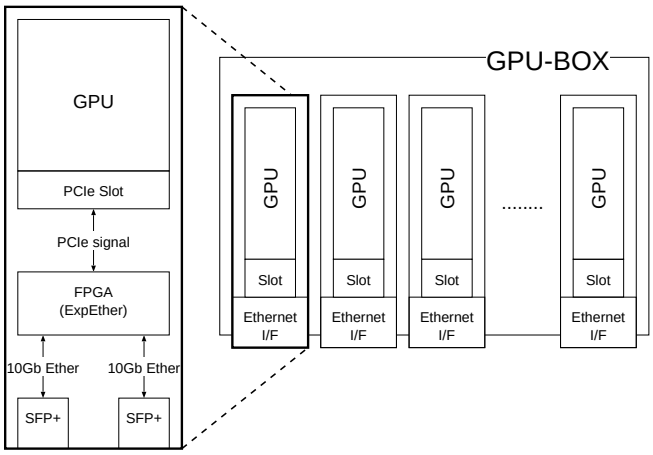


Fig. 4: Overview of GPU-BOX

Table 1: Evaluation Environment

CPU	Intel Core i7 (2.67 GHz)
GPU	NVIDIA Tesla C2050 x6
Host Memory	16 GB
OS	Scientific Linux 6.0
Host Compiler	gcc4.4
CUDA	Toolkit 4.0
Network	10Gb Ethernet x2
Switch	Fulcrum Microsystems Monaco

Table 2: Problem Size of Particle Motion

Number	Particles	Steps
1	1x1024x1024	100
2	1x1024x1024	1000
3	10x1024x1024	1000
4	10x1024x1024	2000
5	10x1024x1024	4000

4.2 GPU-BOX

Here, multi-GPUs with ExpEther is implemented in GPU-BOX which provides PCIe ports and power supply together with the function of ExpEther for extending PCIe interface of GPUs to Ethernet network.

The target GPU-BOX in this paper has slots for eight GPUs. That is, it provides the environment of eight GPUs in term of PCIe slots, Ethernet interface, power supply and space. The power supply of the GPU-BOX is up to 3000W. The Ethernet interfaces are two SFP+ interfaces per a GPU, thus 20 Gbps bandwidth is available in total. Each PCIe in GPU-BOX is extended to Ethernet by ExpEther NIC implemented on FPGAs.

5. Evaluation

In this section, we evaluate the performance of the multi-GPU system with GPU-BOX by executing programs in CUDA.

5.1 Experimental Environment

Table 1 shows the multi-GPU environment used in the evaluation. The evaluated environment uses six GPUs, each of which is NVIDIA Tesla C2050, while the number of slots in the target GPU-BOX is eight.

5.2 Application

For performance evaluation, we implemented two application programs, the simulation of particles motion and the calculation of Advection term.

One has no communication between devices, while the other needs a considerable amount of communication.

These application programs are mainly consisting of following three parts;

- computing in GPUs,
- data transfer between host and GPU, and
- data exchange between GPUs.

The calculation of particle has only two parts, computing and data transfer between host and GPU, while the calculation of Advection term includes all of them.

5.2.1 Simulation of Particle Motion by the Runge-Kutta Method

This application simulates particle motion when initial particle distribution and velocity field are given and there is no interference between particles. It divides time into several steps, and on each step, each particle position is updated with the velocity field by the Runge-Kutta method. For the motion of each particle is independent from another, it is possible to perform every particle motion computation in parallel, and there is no communication between GPUs in multi-GPU environment.

Here, five combinations of parameters are executed for evaluation. Table 2 shows the combinations of problem size and steps.

5.2.2 Calculation of Advection term by Cubic Lagrange Interpolation

Calculation for Advection term of Cartesian grid method is a kind of fluid dynamics computation. It simulates the movement of ink when initial concentration, distribution, and velocity field are given. On this calculation, it separates the entire surface into grid and updates each value of the grid using values of the surrounding grids in a certain time step. On each step, updating of the grid value is independent from other computations. However, in case of computing with

Table 3: Problem Size of Advection Term

Number	X	Y	Steps
1	256	256	1024
2	1024	1024	10280
3	4096	2048	10280

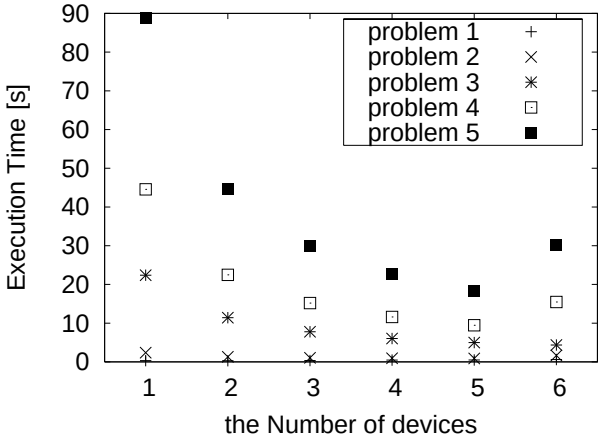


Fig. 5: Execution Time of Calculation of Particle Motion

multiple GPUs, we have to exchange data around memory-boundary between GPUs. Problem size of Cartesian grid method on this evaluation is shown in Table 3.

5.3 Performance versus the number of GPUs

5.3.1 Simulation of Particle Motion

Fig. 5 shows relationship between execution time and the number of GPUs when the particle motion simulation is executed. It is found that the execution time decreases as increasing the number of GPUs. However the execution of the case with six GPUs in the GPU-BOX spent larger time than one with five GPUs. That is caused by the EFE protocol tuning which is not adequate for the system with more than five GPUs.

Fig. 5 shows the performance speedup of the different number of GPUs over a single device. Respectively, the multi-GPU system provided speedup of 1.99, 2.96, 3.92, 4.83 and 5.14 times at most for execution on two to six devices. While performance speedup is directly proportional to the number of devices roughly in case of large problem size, there are cases to get no performance improvement by increasing the number of GPUs. It is mainly caused by the overhead of data transfer between the host processor.

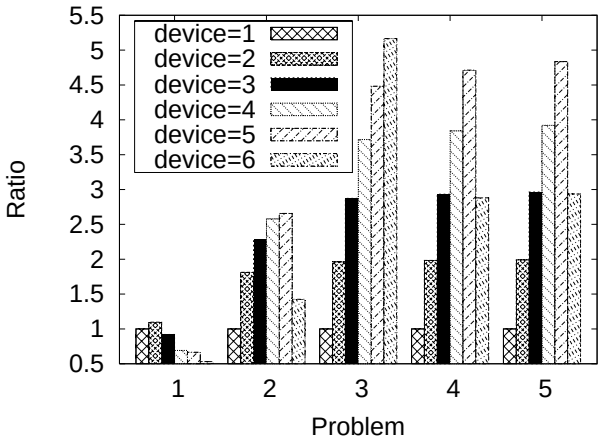


Fig. 6: Performance rate of Calculation of Particle Motion

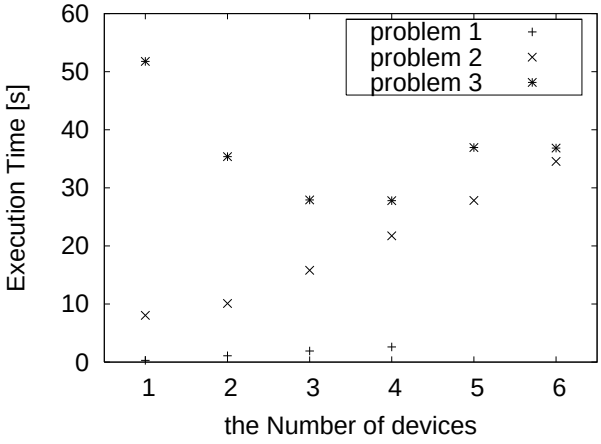


Fig. 7: Execution Time of Calculation of Advection term

5.3.2 Calculation of Advection term

Fig. 7 shows relationship between execution time and the number of GPUs when the Advection term is calculated. In the problem 1, the cases with five and six devices can't be evaluated, since the problem size is so small that there is no data allocated to fifth and sixth device in this application program.

Fig. 8 shows the performance enhancement of the different number of GPUs over a single one. Respectively, the multi-GPU system achieved speedup of 1.45, 1.85, 1.86, 1.40 and 1.41 times in problem 3 for execution with two to six devices. On the other hand, we can see performance degradation on calculating small size problems; problem 1 and 2. The performance degradation is caused by the frequently executed data exchange part for the calculation of

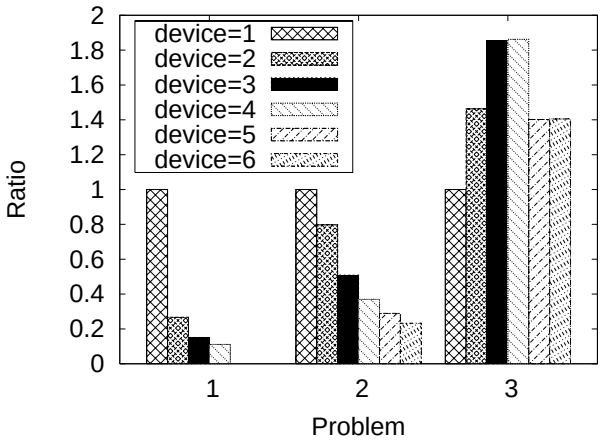


Fig. 8: Performance Rate of Calculation of Advection term

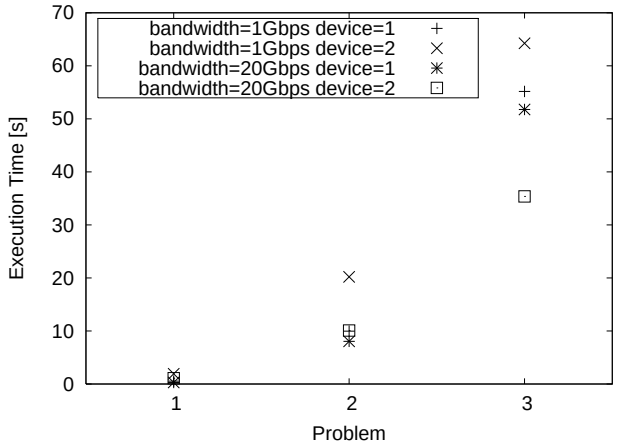


Fig. 9: Relationship between Network and Execution Time

Table 4: Data Transfer Bandwidth in CUDA

Ethernet	1 Gbps	20 Gbps
Host to Device	0.320 Gbps	7.66 Gbps
Device to Host	0.445 Gbps	9.86 Gbps
Device to Device	0.355 Gbps	7.92 Gbps

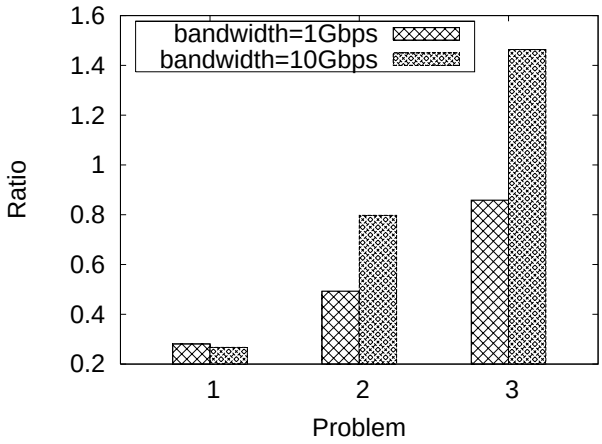


Fig. 10: Relationship between Network and Performance Improvement

Advection term. The data exchange speed mainly depends on the bandwidth of the network, while the exchanged data amount in this application is proportional to the number of GPUs. Although it is difficult to enhance performance in such small problems, their execution time is not so large and the effect of applying multi-GPU system is originally limited. Improving the performance of Ethernet will stretch the target which can be accelerated to smaller size problems.

5.4 Influence of the Network Performance

In this subsection, we evaluate the influence of the network equipment in the GPU-BOX. Table 4 shows the bandwidth of data transfer from the host to the GPU, from the GPU to the host, and between GPUs in CUDA by using Ethernet 1 Gbps and 20 Gbps.

5.4.1 Influence to Application performance

Fig. 9 shows performance of Advection term calculation the case when 20 Gbps and 1 Gbps Ethernet are used with a single and two GPUs. When a single GPU is used for calculation, the difference of the execution time with 1 Gbps and 20 Gbps Ethernet is small. However, 1 Gbps Ethernet increases the execution time on calculating with two GPUs, while 20 Gbps Ethernet can decrease it.

Fig. 10 shows the speedup with two GPUs normalized to that with a single GPU. As the problem size becomes larger, the difference of achieved speedup becomes large between two Ethernets, although the multi-GPU system provides more performance in both. As a result, the speedup of the multi-GPU system is 0.86 with 1 Gbps network, and 1.45 with 20 Gbps network. These results indicate that the multi-GPU system with GPU-BOX obtains benefits of developing Ethernet technology, and GPU-BOX enables user to select network construction according to the application.

6. Conclusion

In this study, we proposed and evaluated multi-GPU system with ExpEther. It allows to interconnect a single host PC and multiple GPU devices with ExpEther which extends PCIe interface to Ethernet.

For evaluating the multi-GPU system, two application programs are used. First, we evaluated performance on the different number of devices. For the program without inter-GPU communication, a system with six GPUs achieved 5.14 times performance as that with a GPU.

On executing the application including data exchange between GPUs, the largest performance improvement was 1.86 times with four GPUs. Then, we evaluated performance using the different networks. From the results, it appears that the bandwidth of network used in GPU-BOX greatly affects the performance of the multi-GPU system.

The following is a list of future work:

- The performance of multi-GPU system with ExpEther must be compared with conventional multi-GPU clusters,
- The performance must be evaluated with the other applications and more number of GPU.
- A larger system which uses multiple switches must be evaluated.

References

[1] NVIDIA, "NVIDIA CUDA Compute Unified Device Architecture," http://developer.nvidia.com/object/cuda.html.

[2] A. M. Devices, "Ati stream sdk getting started guide (v2.3)," http://developer.amd.com/GPU/ATISTREAMSDK/DOCUMENTATION/Pages/default.aspx.

[3] NVIDIA, "The OpenCL Specification Version: 1.0," 09.

[4] T. I. of Technology Global Scientific Information and C. Center, "Tsubame2," http://www.gsic.titech.ac.jp/tsubame2.

[5] A. Shitara, T. Nakahama, M. Yamada, T. Kamata, Y. Nishikawa, M. Yoshimi, and H. Amano, "Vegeta: An implementation and evaluation of development-support middleware on multiple opencl platform," in *The Second International Conference on Networking and Computing*, November 30 - December 2, 2011.

[6] R. Aoki, S. Oikawa, T. Nakamura, and S. Miki, "Hybrid opencl: Enhancing opencl for distributed processing," in *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, 26-28 May 2011, pp. pp.149 – 154.

[7] Integrated Device Technology, "Pci express switches," http://www.idt.com/products/interface-connectivity/pci-express-solutions/pci-express-switches.

[8] PCI-SIG, "Pci express external cable 1.0 specification."

[9] CONTEC, "PCI Express External Cabling (PCISIG) Compliant Expansion Units," http://www.contec.com/products/bus_exp/pcie.php.

[10] T. Miyoshi, H. Irie, K. Shima, H. Honda, M. Kondo, and T. Yoshinaga, "Flat: a gpu programming framework to provide embedded mpi," in *GPGPU-5 Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, 2012, pp. pp. 20–29.

[11] J. Suzuki, Y. Hidaka, J. Higuchi, T. Yoshikawa, and A. Iwata, "Expressether - ethernet-based virtualization technology for reconfigurable hardware platform," in *High Performance Interconnects*, 2006, pp. pp.45–51.

[12] NEC Corporation, http://www.nec.co.jp.

Performance Analysis on Several GPU Architectures of an Algorithm for Noise Removal

M. G. Sánchez[1], V. Vidal[2], J. Bataller[2] and G. Verdú[3]
[1]Department of Systems and Computing, Instituto Tecnológico de Cd. Guzmán, Cd. Guzmán, Jalisco, Mexico
[2]Department of Informatics Systems and Computing, Universitat Politècnica de València, Valencia, Spain
[3]Department of Chemical and Nuclear Engineering, Universitat Politècnica de València, Valencia, Spain

Abstract - *In this paper, we present an efficient implementation of parallel algorithms to remove noise in digital images using different Graphics Processing Units (GPUs). The algorithm, based on the concept of peer group, uses a fuzzy metric for finding wrong pixels and the Arithmetic Mean Filter (AMF) to correct it. There are many factors to study in order to get an optimal implementation of an algorithm on a GPU. Our algorithm has been implemented with two different approaches to access the data: Shared and Texture memory. Also, the number of threads and its arrangement have been studied. The test has been conducted on two different cards: Tesla-Fermi and GeForce.*

Keywords: Parallel Algorithm, GPU, Tesla, Noise removal in images.

1 Introduction

Many filters have been introduced in other papers in order to reduce the noise in images. Some works are based on the concept of peer group, the fuzzy metric, Arithmetic Mean Filter (AMF) and Vector Median Filter for the detection and elimination of noise [1], [2]. These types of filters have recently shown good quality results but the execution takes too much time. In this paper, we use the advantages of parallelism offered on the GPU to speed up the process of removing noise in a digital image using CUDA [3].

There are not many jobs reporting accelerated algorithms on GPU to remove noise in an image, therefore some works of image processing using the GPU are presented in [4], [5], [6], [7], [8], [9]. Some irregular data structures such as graphs that involve dynamical memory management are accelerated using Graphical Processing Units (GPUs) and CUDA [10].

The applications running on the GPU should take advantage of the parallel hardware to achieve good performance. The hardware optimizations are considered depending on the application. The application is executed in on different hardware to compare its performance, therefore is necessary to perform the appropriate optimizations in order to take advantage of the available hardware and performance improvement.

The GPU has several types of memory, the way to access them and the computational cost is different for each case. In this paper, we perform an analysis to access the data in GPU memory using two optimizations (Shared memory and Texture memory) in order to find the best type of memory with the best performance.

The Shared memory is expected to be a low-latency memory near each processor core, for this reason we take the advantage to performance optimization applying the implementation of this type of memory.

We present the performance obtained by running the algorithm in the architecture Tesla M2050 (compute capability 2.0), and we compared the results with the performance in the architectures GeForce GT 120 and GeForce 9800 GX2, both with compute capability 1.1. In order to observe the behavior in each architecture for different amount of processed data, we have used an image taken from the Kodak database [11], which we have re-sized and added impulsive noise [3].

We need to analyze the algorithm to know which parts are more appropriate to be parallelized to achieve high efficiency.

The main contributions of this paper are: to identify the parallelism in the algorithm to eliminate noise in an image and the use of the best optimization depending on the GPU architecture and the amount of data to be processed.

This paper is organized as follows: In Section 2, we review the method of peer group and fuzzy metric to eliminate impulsive noise. The relevant features of the CUDA GPU architecture are described in Section 3. In section 4 we expand on details our serial and parallel implementations of the method. We present performance results in Section 5 and, we offer some conclusions and further work in Section 6.

a) b) c)

Figure 1. a) Original Image, b) Noisy Image and c) Filtered image

2 Method Peer Group and Fuzzy Metric (PGFM)

The method to remove noise in digital image uses the concept of peer group and fuzzy metric. It consists of two steps. In the first step (detection), erroneous pixels are detected and corrected in the second step (filtering). In the detection step, the pixels are labeled as corrupted and uncorrupted. Each pixel is analyzed in windows (W) of size nxn (n =3, 5, 7 …), until complete all the pixels of the image. The corrupted pixels are corrected through Arithmetic Mean Filter (AMF) in the second step (filtering step).

The concept of Peer group [2] and the fuzzy metric [3] are used for classification of the pixels. The peer group of the central pixel (x_i) is defined as the set of its neighbour pixels (x_j) which present similar features according to an appropriate distance or similarity measure. Fuzzy metric is an example of metric between two pixels.

The pattern of the fuzzy metric used in this work is:

$$M(x_i, x_j) = \prod_{l=1}^{3} \frac{\min\{x_i(l), x_j(l)\} + k}{\max\{x_i(l), x_j(l)\} + k}, \quad (1)$$

where the value of k is greater than zero and ($x_i(1)$, $x_i(2)$, $x_i(3)$) is the color vector for the pixel x_i in space color RGB.

The pattern of the peer group with this fuzzy metric is:

$$P(x_i, d) = \{x_j \in W \ : M(x_i, x_j) \geq d\}, \quad (2)$$

where $0 \leq d \leq 1$ is the threshold distance.

Implementing a similar process to work [12], we have obtained the optimum values of k, m y d. The values are: m = 3, k = 1024, d = 0.94.

Arithmetic Mean Filter is defined as:

$$AMF_{x_i} = \frac{\sum_{x_j \in W, i \neq j} x_j}{(nxn - 1)} \quad (3)$$

where x_j corresponds to a pixel within the window (W). The AMF is obtained for each RGB color.

Figure 1 shows the original image (512x768) as well as, noisy image and filtered image after running the method PGFM.

3 NVIDIA GPU Architecture and Features

In recent years, the GPUs have become very popular as compute accelerators for non-graphics applications, especially in the field of high-performance computing.

NVIDIA introduced CUDA, a technology which enables these units to be used to develop programs for other calculation purposes, which introduces a few extensions to C++ language.

GPU architecture details are described in [1], but here is a brief summary of the memories and other characteristics of GPU used. This summary will help to explain the parallel implementation to remove noise in an image.

Physically, the GPU, called a device in CUDA terminology, contains a set of multiprocessors. These execute programs following the SIMD (Single Instruction, Multiple Data) model, where by each of the multiprocessor's processor clock cycles executes the same instruction applied to different data, taking into account each application if this instruction is performed by a different thread.

A device has a physical memory that can be used in different ways. The main use is as global shared memory among the GPU multiprocessors. However, this memory also permits its several areas to be used in other modes, as local memory, as Texture and as a constant reading area, shared between all the threads.

Internally, each multiprocessor has four kinds of memories: a set of registers, a read/write cache memory, a constant read-only cache memory, and a read-only cache memory called Texture cache. These memories explain in detail in [9].

The large variety of memories and their different features complicate the task of achieving optimum performance in programs using CUDA.

The design problems of a CUDA program are:

- Deciding the number of threads and their organization in blocks.
- Deciding at each moment the best location (among the different available memories) for the input and output data, and how to access to them, performing the necessary copies at the appropriate time.

The main features of the GPU hardware that we have used in this work are shown in table 1. The compute capability of GeForce GT 120 and GeForce 9800 GX2 is 1.1 and of Tesla M2050 is 2.0 (Fermi architecture).

Bandwidth is one of the most important gating factors for performance. We have maximized the use of the available memory bandwidth for each category of memory. Figure 2 shows the bandwidth and the size of the memory device for each one of the architectures.

The number of threads used per block is therefore restricted by the limited memory resources of a processor core. In NVIDIA Tesla architecture, a thread block contains 1024 threads.

Table 1: Hardware features of GPUs

Model	*Mp*	Global Memory	Shared Memory	CUDA cores
GeForce GT 120	4	512 MB	16 KB	32
GeForce 9800 GX2	16	512 MB	16 KB	128
Tesla M2050	14	3 GB	48 KB	448

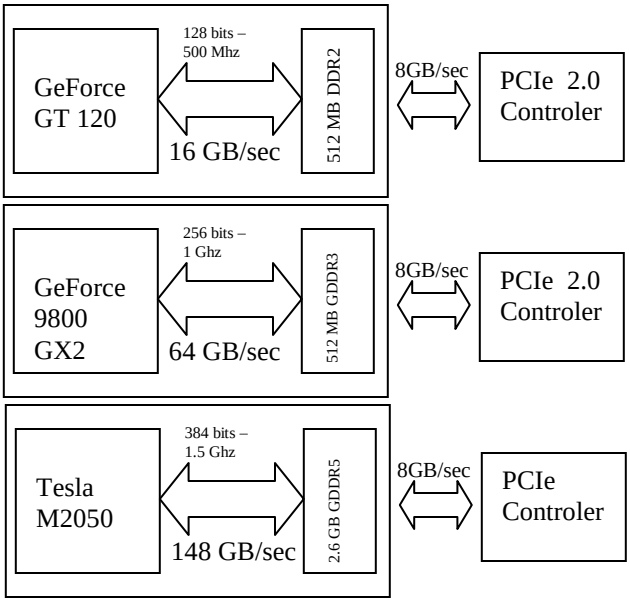


Figure 2. Bandwidth and memory size of the device for three GPU architectures.

The blocks are organized into a one-dimensional or two-dimensional grid of thread blocks. In our case, we handle 2D blocks. The total number of threads is equal to the number of threads per block multiplied by the number of blocks. Each block within the grid can be identified as a one or two-dimensional.

The number of thread blocks in a grid is typically dictated by the size of the data being processed rather than by the number of processors in the system. This number can greatly exceed the number of block threads.

The performed work by a processor in each GPU depends on the blocks launched by the kernel, considering the size of the image. We have handled 2D structures, so the total number of threads per block should be as close as possible to the maximum number of threads per block in each one of used the architectures. The GeForce maximum amount of threads per block is 512, so that the threads building blocks of 16x16 for a total of 256 threads, thus there will be two blocks running simultaneously on each processor. In the Tesla, the maximum number of threads per block is 1024, so we can make blocks of 32x32 for a total of 1024 or 16x16 for a total of 512 threads in each block.

4 Algorithm to remove noise in digital images

In this section, we describe the sequential CPU implementation and parallel GPU implementation of the proposed method in section 2.

4.1 The Sequential CPU Implementation

Algorithm 1 shows the sequential implementation on the CPU to remove noise in an image. This Algorithm is a modification of [2]. It is divided into two phases: detection and filtering.

Algorithm 1: Detection and Filtering using the Peer Group Filter (PGF). Sequential Implementation.

Input: pixels of the image, *m*, *k*, *d*, *n=3*.
Output: Filtered Image.
//Detection
for each pixel x_i of the image **do**
  Building *W* of size *n* x *n* centered in x_i
  if ($M(x_i,x_j) \geq d$) **then**
    $x_j \in P(x_i,d)$;
  end if
  if ($\#^* P(x_i,d) \geq (m+1)$) **then**
    x_i *is declared as uncorrupted pixel;*
  else
    x_i *is declared as corrupted pixel*;
  end if
end for
//Filtering
for each *corrupted pixel* x_i **do**
  Building *W* centered in x_i;
  Calculate the AMF of uncorrupted pixels of the *W ;*
  Replace x_i with the output of the AMF;
end for

* (#) = cardinality

4.2 Implementation on GPU

In order to speed up the processing of the algorithm 1 described in the previous subsection, we have made a parallel implementation on the GPU. The algorithm has been modified to work in parallel. We have done the detection step on the GPU through a kernel (detection kernel). Another kernel (filtering kernel) was defined to run filtering step.

In order to obtain a good performance in applications, one of the factors to be considered is the number of threads and blocks in each launched kernel. This factor depends on the maximum number of threads supported by the graphics card and the size of data to consider.

In this work, we calculate the number of blocks that are released into the kernel depending on the size of the data. After memory space is allocated for the GPU to store the

image, data are transferred from RAM-Host to RAM-GPU. Once the data are in graphics card then the detection kernel and filtering kernel are launched. After launching the detection and filtering kernel, the threads are synchronized. Once the filtering kernel has finished, the data is transferred from RAM-GPU to RAM-Host. These steps are described in pseudo-code in Algorithm 2.

We used 2D linear memory allocation in the memory device to store data. To speed up image transfer between host and device memory we use page-locked. Device memory can be allocated as either linear memory or as CUDA arrays. We have used for this application linear memory.

The amount of threads to launch is the same number of pixels as the image has. Each thread corresponds to one pixel of the image and it builds its window of size *n*x*n*. The thread classifies as corrupt or not corrupt the central pixel of the window, ie marks the pixel that is analyzed while the neighbours are analyzed simultaneously by other threads. We reserve one byte for each pixel which we will call it padding (*p*) in order to assign the state of the pixel (corrupt or not corrupt), one pixel is composed of 4 bytes (RGB*p*). All the pixels after detection kernel are labeled as corrupt or not corrupt. For the filtering kernel, we launched the same amount of threads as in the first step. Some threads will not work because they do not correspond to pixels labeled as corrupted. Considering that the computational cost of the memory transfers between RAM-Host and RAM-GPU is high, we prefer to launch the same amount of threads that are released in the detection kernel.

Algorithm 2: Pseudocode for the parallel GPU implementation.

```
Allocate device memory for the pixels;
Data transfer from Host memory to Device memory;
do in parallel on the device using T threads:
    call Detection_kernel
    Synchronization of threads;
do in parallel on the device using T threads:
    call Filtering_kernel
    Synchronization of threads;
Data transfer from Device memory to Host memory;
Free device memory;
```

The detection kernel is described in Algorithm 3. In this algorithm the called *fuzzy_metric_function* (a_pix, b_pix), is the function to calculate the fuzzy metric between central and neighbour pixels of the window. *a_pix* is the central pixel and *b_pix* is a neighbour pixel.

The filtering kernel is described in Algorithm 4. In this algorithm the called *mean_function* (b_pixOK), is the function to calculate the AMF, described in equation 3. *b_pixOK* is an array of the uncorrupted pixels.

When using Shared memory, the processes that are performed in each kernel, are described in algorithms 3 and 4 with the following changes :

Sentence 4 of algorithm 3 and sentence 5 of algorithm 4, is replaced by:

a_pix← load from device memory to Shared memory RGB value of x_i from row and col;
Synchronizes with all the other threads of the block;

Sentence 7 of Algorithms 3 and 4, is replaced by :
b_pix← load from device memory to Shared memory RGB value of neighbour in (i,j) from row and col;
Synchronizes with all the other threads of the block;

Algorithm 3: The Detection_kernel

Input: pixels of the image, *m*, *k*, *d*, *n=3*.
Output: pixels labeled as corrupted or uncorrupted.
For each thread that is associated with a pixel x_i:

```
1: col ← global thread ID in col;
2: row ← global thread ID in row;
3: byte padding (p) is initialized to zeros;
4: a_pix ←load RGB value of x_i from row and col;
5: for i←-1 to 1 do
6:    for j ←-1 to 1 do
7:       b_pix ← load RGB value of neighbour in (i,j)
                 from row and col.
8:       dist ←load the distance obtained after of
                 call fuzzy_metric_function (a_pix, b_pix)
9:       if dist ≥ d then
10:          //pixel (b_pix) ϵ P(a_pix,d)
11:          cardinality ← cardinality+1;
12:      end if
13:   end for
14: end for
15: if cardinality < (m+1) then
16:    //pixel is declared as corrupted;
17:    c_pix ← load in four byte (p) a value of 1;
end if
```

Fuzzy metric and peer group are calculated reading the data of Shared memory.

A texture can be any region of linear memory or a CUDA array, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations. When we use Texture memory, we declare a texture to access the data. The texture is used in both detection and filtering kernel.

Sentence 4 of algorithm 3 and sentence 5 of algorithm 4, is replaced by:

a_pix← load from Texture memory RGB value of x_i from row and col ;

Sentence 7 of Algorithms 3 and 4, is replaced by:
b_pix← load from Texture memory RGB value of x_i from row and col.

Algorithm 4: The Filtering_kernel

Input: pixels of the image labeled as corrupted, n=3.
Output: Filtered Image.
For each thread that is associated with a corrupted pixel x_i:
1: col ← global thread ID in col;
2: row ← global thread ID in row;
3: a_pix ← load RGB value of x_i from row and col;
4: **if** (a_pix in byte p) ==1 **then**
5: **for** i ←-1 to 1 **do**
6: **for** j ←-1 to 1 **do**
7: b_pix ← load RGB value of neighbour in (i,j) from row and col.
8: **if** (b_pix in byte p) == 0 **then**
9: b_pixOK ← load b_pix
10: **end if**
11: **end for**
12: **end for**
13: mean ← load the media obtained after of
14: **call** mean_function (b_pixOK)
15: a_pix ← load the mean value;
16: **end if**

5 Performance Evaluation

5.1 Experimental Study

In the experiments, we used one image obtained from [10] which we have re-sized and added 10% of impulsive noise with *imnoise* instruction of MATLAB. Table 2 summarizes the sizes of the images that we used.

Table 3 shows the blocks configuration, threads and occupation of the cores in each one of the GPUs used. The number of blocks per core is obtained by dividing the total number of blocks by the CUDA cores available (table 1). For the Tesla architecture the number of blocks created for the first image size and 32x32 threads per block is 96 blocks and CUDA cores are 448, ie, 352 cores are inactive. For the second image size, 64 cores are inactive. From the 3rd. image size, all cores have to work. The algorithm 2 was implemented for each size image in order to eliminate the noise in the image. We use three architectures (GeForce GT 120, GeForce 9800 GX2 and Tesla M2050) with different characteristics.

Table 2. Size image used in the experiments.

Case	*Image Size*
1	256x384
2	512x768
3	1536x1024
4	3072x2048
5	6144x4096

Table 3. Configuring blocks and grids.

	Image size	Max. Threads per block	Threads per block	Number of blocks	Number of blocks per core
GeForce GT 120	1	512	16x16	16x24=384	12
	2			32x48=768	24
	3			64x96=1536	48
	4			128x192=3092	96
	5			256x384=6144	192
GeForce 9800 GX2	1	512		16x24=384	3
	2			32x48=768	6
	3			64x96=1536	12
	4			128x192=3092	24
	5			256x384=6144	48
Tesla M2050	1	1024		16x24=384	1
	2			32x48=768	1
	3			64x96=1536	3
	4			128x192=3092	6
	5			256x384=6144	13
	1	1024	32x32	8x12=96	1
	2			16x24=384	1
	3			32x48=1536	3
	4			64x96=6144	13
	5			128x192=24576	54

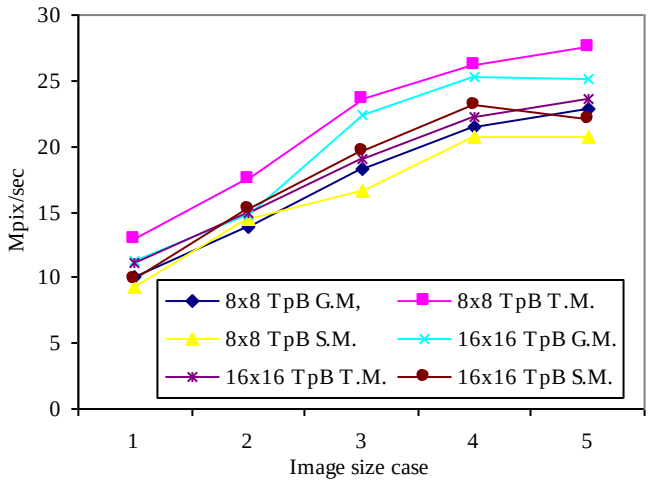


Figure 3. Performance Mpix/sec when using block size 8x8, 16x16 and 32x32 with diffrent image size on GeForce GT 120.

5.2 Results and Analysis

In this section we evaluate the GPU performance when running the noise elimination algorithm.

The first experiment was to study the performance on the GPU with two possible combinations of block size (8x8 and 16x16) in GeForce GT 120 and (8x8, 16x16 and 32x32) in M2050.

The results of the GeForce GT 120 architecture are shown in figure 3. We can see that the best block size is 8x8 with Texture memory and 16x16 for the other two accesses. Shared memory with 16x16 blocks is best than texture memory with 16x16 blocks and global memory with 8x8 blocks. The reason is that the processor is inactive, waiting for

more data, in other words, the relationship between the processor clock speed and the bandwidth are not appropriate. The results are the same in GeForce 9800.

The results of Tesla architecture using the two forms of access (directly to Global memory or through Texture memory) are shown in figure 4. We can see that when using the Global memory the best block size setting is 16x16 and the worse is 8x8. It obtains more Mpix/sec when accessing data through the texture memory. For the largest size that we have tested (larger than 512x768) and 32x32 threads per block, we can see that the results of using Global memory or Texture memory have very good behavior.

In the rest of the paper, the results are shown with 16x16 block size except for texture memory in the GeForce GT 120 and GeForce 9800 GX2, in this case is used 8x8 block size.

Figure 5 compares the performance of Mpix/sec obtained in the three architectures of GPU used and the three types of access to data in global memory.

The performance of the GeForce GT 120 is better when Texture memory is used compared to the other two forms. We also note that in the image with size 6144x4096 the performance with Shared memory decreases, due to the overhead associated with the transfers. All the threads gets data from Global memory to Shared memory.

The performance obtained using the parallelized algorithm on GeForce9800 GX2 with the Texture memory is the best option to access the data followed by Shared memory. In image sizes 6144x4096 with Global memory and Texture memory the performance decreases, due to the overhead generated by the transfers. In this case the processor clock speed with the bandwidth is tightly coupled; this is reflected in the difference between global memory and Texture.

Texture memory is also the access that provides better performance when using M2050 architecture. The use of Shared memory access and the Global memory access provides very similar results. We can see that in Tesla M2050, regardless of the type of access, the performance of the image size 512x768 is lower than other image sizes. This is because there are insufficient data to allow cores to work.

We also see that the Tesla-Fermi architecture outperforms compared with the other two GPU architectures at all image sizes. If there is more bandwidth, the accesses are faster. For the three architectures the performance is better with Texture memory for any image size.

With these results, we can say that using the Texture memory is the best option for all ways of access to data. Shared memory architectures should be used in architectures with similar features to the GeForce 9800 GX2.

Figure 6 shows that the processing time on GPU depends on the percentage of noise in the image. As we can see, when the image has more noise, processing time is greater. The increase is in the filtering step; in the detection step the processing time is independent of the percentage of noise. A quality study of this parallel algorithm has been analyzed in work [9]. In that paper, we showed that the quality achieved by implementing the algorithm is competitive compared with other algorithms.

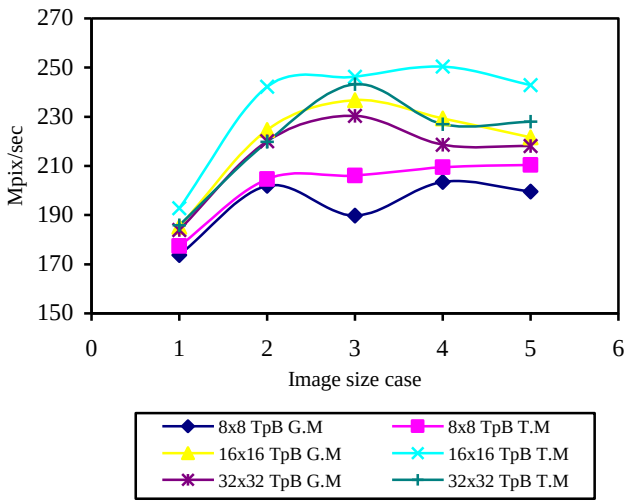


Figure 4. Performance of Mpix/sec. when using block size 8x8, 16x16 and 32x32 with different image size on Tesla.

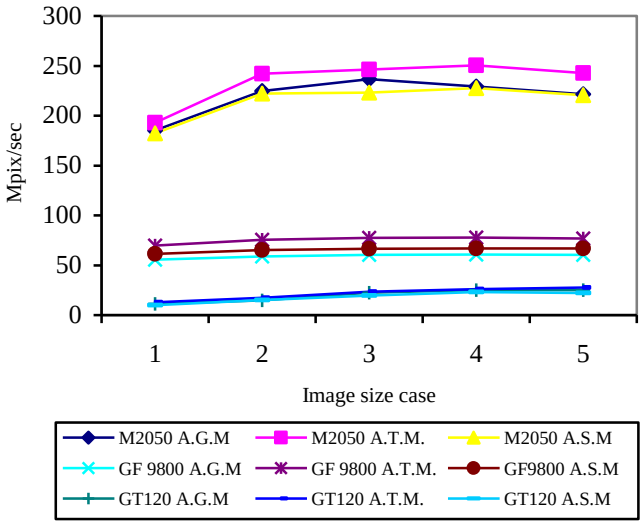


Figure 5. Performance in the three models of GPU and the three ways to access the data.

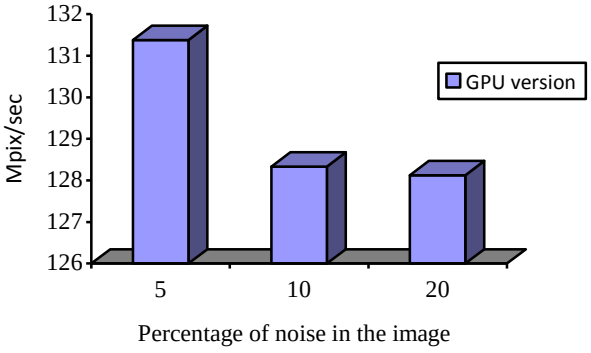


Figure 6. Mpix/sec processed for different percentage of noise.

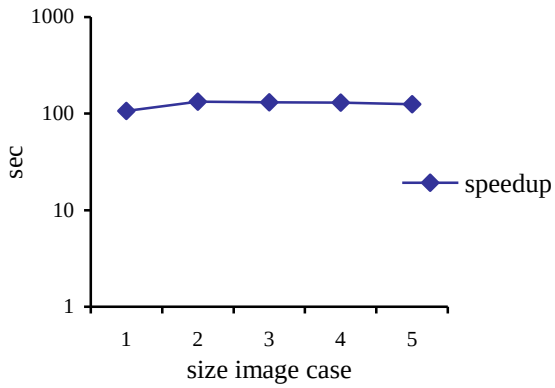


Figure 7. GPU vs CPU speedup for different image sizes.

Finally, we report in Figure 7, the speedup of the sequential version of the algorithm running on the CPU and GPU-parallel version using Texture memory and Tesla architecture. As noted, in the worst case the GPU version (parallel) is 106 times faster than sequential, which is an excellent result, showing that using the GPU is the best for noise correction.

6 Conclusions

In this paper, we present efficient implementations of parallel algorithm to remove noise in digital images using different Graphics Processing Units. The algorithm uses the concept of peer group, the fuzzy and Arithmetic Mean Filter (AMF) to detect and remove noise. When accessing to the pixels of the global memory, we have considered two ways to optimize the access to them; one is through Shared memory and the other through Texture memory. We have presented an analysis of the performance of these two models of graphics cards (GeForce vs Tesla-Fermi) with different forms of access and different image sizes. We have performed an analysis of optimal block size for these two architectures. Tesla architecture supports a bigger number of threads per block, being 256 threads per block the best size. The results show that using the Texture memory is the best. Global memory is not an option for reading the data from the GPU memory for this application and with GPU architectures used. In the Fermi-Tesla architecture we process 165-215 Mpix/sec higher than the GeForce, running the algorithm to eliminate noise in an image with the largest tested size. In a future work we will see, if this behavior is similar to other kinds of architecture with similar hardware characteristics to this application.

Acknowledgment

This work was funded by the Spanish Ministry of Science and Innovation (Project TIN2011-26254) and Ma. Gpe.would also like to acknowledge DGEST ITCG for the scholarshipawarded through the PROMEP program (Mexico).

7 References

[1] Camarena, J-G., Gregori, V., Morillas, S., Sapena,A., "Two-step fuzzy logic-based method for impulse noise detection in colour images", Pattern Recognition Letters 31, pp.1842-1849, 2010

[2] Camarena, J-G., Gregori,V., Morillas, S., Sapena, A., "Some improvements for image filtering using peer group techniques", Image Vis. Comput 28 No. 1, pp.188-201, 2010.

[3] http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf

[4] Sánchez. M.G., Vidal, V., Bataller, J., Arnal, J., "A Fuzzy Metric in GPUs: Fast and Eficient Method for the Impulsive Image Noise Removal", in Proc. ISCIS 2011.

[5] Stone, S. S., Haldar J.P.,Tsao S.C.,Hwu W.W., Liang, Z-P., Sutton B.P., "Accelerating Advanced MRI Reconstructions on GPUs," in Proceedings of the 5th International Conference on Computing Frontiers, May5-7, 2008.

[6] Xu, W., Mueller D., "Learning Effective Parameter Settings for Iterative CT Reconstruction Algorithms," in Fully 3D Image Reconstruction in Radiology and Nuclear Medicine Conference, 2009.

[7] Li, L., Li, X., Tan, G., Chen M., Zhang P., "Experience of Parallelizing cryo-EM 3D Reconstruction ona CPU-GPU Heterogeneous System," in Proceeding HPDC 11 Proceedings of the 20th international symposium on High performance distributed computing, 2011.

[8] Anderson R.F., Kirtzic J.S., Daescu, O., "Applying Parallel Design Techniques to Template Matching with GPUs", in Springer-Verlag New York Inc, Volume: 6449, 2011.

[9] Sánchez. M.G., Vidal, V.,, Bataller, J., Arnal, J., "Implementing a GPU fuzzy filter for Impulsive Image Noise Correction", in Proc. CMMSE 2010.

[10] Leist, A., Hawick, K.A., "Graph Generation on GPU using Dynamic Memory Allocation", in Technical Report CSTN-126. May 2011.

[11] Kodak, http://r0k.us/graphics/kodak/index.html

[12] Morillas, S., Gregori V., P.Fajarnés, G., "Isolating impulsive noise pixels in color images by peer group techniques", Computer Vision and Image, 2008.

GPU Computing and CUDA technology used to accelerate a mesh generator application

Adriana Gaudiani[1], **Santiago Montiel**[1], **and Javier Pimás**[1]
[1]Instituto de Ciencias, Universidad Nacional de General Sarmiento
San Miguel, Buenos Aires, Argentina

Abstract— *The potential of GPU computing used in general purpose parallel programming has been amply shown. These massively parallel many-core multiprocessors are available to any users in every PCs, notebook, game console or workstation. In this work, we present the parallel version of a mesh-generating algorithm and its execution time reduction by using off-the-shelf GPU technology. We use commodities GPUs as a useful CPU co-processor to improve this kind of applications, characterized by a high level of data parallelism. Compared to the sequential algorithm, our techniques achieve 6X overall performance for GPU-CPU implementation; furthermore we achieve 50X speedup when implementing core operations of the algorithm. Results show that GPU provides a helpful platform for high performance computing to improve the execution time of these applications.*

Keywords: Parallel algorithms, Multicore processors, Graphics processing units.

1. Introduction

In the past few years, Graphics Processing Units Computing (*GPUs*) has demonstrated to provide an increased performance computing architecture for applications which can be written to take advantage of many core GPUs. The idea to use the GPU for general purpose computations starts about 2003. However, it was reserved only for specialist developers in graphic rendering. It was possible since 2007, when NVIDIA released the software technology called Compute Unified Device Architecture (CUDA) simultaneously with its TESLA Architecture [1] [2]. Since then, GPUs have increased in capability and programmability and have gained wide popularity among research community [3]. Owens et al. widely explain why GPUs increased faster than CPUs. Furthermore, this technology is available, inexpensive and can be found in off-the-shelf graphics cards for PCs. CUDA is designed for extended standard C/C++ code with GPUs parallel features and it provides a unified computing platform to take advantage of the GPUs power and to leverage general purpose parallel applications [4].

In this paper, we present as we used cutting-edge computing resources, such as GPUs multicores and CUDA, to accelerate the execution time of a finite element mesh generator algorithm, achieving a significant parallel speedup.

Mesh generation is a key step in many scientific computations and computer graphics. It had its origin in the 50's with structural analysis problems. Finite element is a numerical method used to solve partial differential equations approximately; whose first step is mesh generation.

We use GPUs as a floating point parallel CPU coprocessor to improve the mesh generation algorithm *Distmesh*, created by P. Persson and G. Strang. Distmesh authors wrote an efficient Matlab algorithm to provide a simple code to produce high quality meshes. We wrote a parallel version for this algorithm to introduce an interesting general purpose application for GPU computing.

Organization: The paper is structured as follows. In section 2 we present a general view of Persson-Strang mesh generator algorithm and an overview of architectural features of GPUs. In section 3 we present our sequential version of Distmesh algorithm and highlight its features. Section 4 describes the design and implementation of our GPU version of mesh generator algorithm. Section 5 gives experimental results. In section 6 we briefly introduce related works and section 7 gives conclusions.

2. Background

In this section we present important background concepts, relevant to this paper. First, we present a briefly description of Persson-Strang Algorithm. Then, we outline some important issues for using GPU.

2.1 Persson-Strang Algorithm

Per-Olof Persson and Gilbert Strang developed a simple and public mesh generator code for Matlab, called Distmesh [5]. They offer an iterative technique based on a physical analogy between a simple mesh and a trust structure, combining a signed distance function and forces movement at each node. The results obtained are high quality meshes.

Many problems are defined on irregularly shaped domains, so unstructured meshes, far better than structured meshes, can be flexibly tailored to the physics of these problems. The problem that arises is the complex, and nearly inaccessible, meshing software code. We have chosen Persson and Strang algorithm because of its simplicity and accuracy. We present below a brief description of the algorithm.

Initial nodes position may be chosen by equally spaced distribution, and this works well for simple geometries. The user can define a function *h* to set mesh resolution. This function *h(x,y)*, in 2D, is used to refine complex geometry mesh and thus achieve geometrical adaptivity; it needs to be resolved by small elements. The meshpoints define the truss structure and a Delaunay triangulation algorithm determines the topology. Delaunay method set triangles in 2D, or tetrahedra in 3D, to fill the convex hull of the input domain mesh points [6]. In response to the mechanical analogy, triangle edges correspond to bars (or springs) and mesh points correspond to truss joints. The numerical method assumes that a displacement force is exerted on the bars. In every iteration, the new location of the points is obtained by calculating a force of static equilibrium. Delaunay triangulation is needed whenever the points are separated far; thus adjust the topology. For a very detailed explanation on the Persson method, the interested reader should consult [7].

2.2 GPU and CUDA overview

GPU is a massively multi-threaded multiprocessor architecture and its data level concurrency stand out. The threads are organized in two-level hierarchy. The lower level is a *block*, which contains a large number of threads. The higher level is a *grid*, which consists of a group of blocks. The maximum dimension of blocks and grids is determined by the GPU architecture. Parallel threads share memory and synchronize using barriers [8], [9].

The key to effectively using GPU is to understand its memory hierarchy, which consists of three levels of memory. Programmers can explicitly manage data stored in them. *Device memory* is the global GPU memory which is accessible from all the threads. *Shared memory* is an on-chip memory. It's a low latency memory shared by all the threads within a block. *Texture and constant memory* are used to store explicitly declared read only data.[10].

NVIDIA's CUDA enables to divide the parallel program execution in tasks that can run across thousands of concurrent threads, over hundreds of processor cores. This programming model is known as a Single-Program Multiple-Data (SPMD) and it allows to program GPUs for general purpose. Tesla GPU, the NVIDIA device used for our experiences, manage efficiently a huge sum of threads employing a *Single-Instruction Multiple-Thread* (SIMT) parallel programming architecture. The task performed by every thread is managed writing special functions, called *kernels*. The kernel task is mapped over a set of threads, representing the work to be done at a simple point in the domain. We wrote the kernels using C and CUDA, an extended version of C; in this way, we mapped the kernels on the GPU manycore processors. For a more detailed description of CUDA, GPU architecture and Tesla architecture, you can refer to [10][11][12].

3. General features of sequential version

We first implemented a C++ sequential code of Distmesh algorithm. We took into account some important items in the original version of the mesh generator algorithm, as we describe below.

- A distance function d determines the domain geometry by means of a signed distance, which is negative inside the region. It was an essential decision, as authors remark. This function is calculated at a meshpoint set and also for calculating nodes distance to the closest boundary point.
- This implementation uses a linear function for repulsive forces, but it does not allow attractive forces.

$$f(l, l_0) = \begin{cases} k(l - l_0) & \text{if } l < l_0 \\ 0 & \text{if } l \geq l_0 \end{cases}$$

- The resultant force *FTot* is the sum of all force vectors meeting at a mesh node. Each bar exerts a force $f(l, l_0)$ depending on its actual length l and its relax length l_0.
- The relax length l_0 is constant for uniform meshes and it's required $f = 0$ for $l = l_0$. Distmesh authors choose l_0 slightly larger than the length desired -20% is a good rate- This calculation depends on the total sum of bars length.
- The time step for Euler method is Δt parameter. The parameter *geps* is used to calculate the tolerance in geometry evaluations, and it's used to decide whether a point is outside.
- All points going outside the domain during the update, p_n to p_{n+1}, are projected back to the boundary. The numerical gradient of d gives the direction of the point movement.

The following are the key steps of our sequential code. In general terms, these steps correspond to those of Distmesh. Although, we modify the original data structures for better performance of the sequential algorithm. We will refer to this later.

Data Input: As a first step, we create a uniform distribution of mesh points within the input desired geometry. These points are the mesh-nodes. The resulting mesh points are regularly placed at a distance h_0 from their closest neighbors.

Triangulation: An important step in this algorithm is Delaunay triangulation. At every iteration, we compare the actual points positions with that of the previous triangulation. When the maximum displacement is greater than a predefined tolerance, a Delaunay retriangulation determines the new meshpoints set replacing the old ones, in order to guarantee Delaunay properties.

Update: The bars lengths are used to calculate the bar forces components. The resultant node force is the sum of the force vectors, from all bars meeting at a node. This result contributes to update node positions.

Projection: The update process may place some points outside the geometry. Once these points are found, they are

projected back to the boundary, in response to a normal force. We use the numerical gradient of the distance function to calculate their move direction to the closest boundary point.

Our first CPU sequential code was a translation of Matlab Distmesh algorithm. Distmesh is an efficient algorithm for Matlab, it's completely vectorized to avoid loops. The authors use a sparse matrix to compute mesh-nodes movement. The sparse matrix dimension is determined by total points (n) and bars (m) of the mesh. Distmesh code uses a Delaunay Matlab function in order to determine truss topology. We selected the Delaunay C-code written by Geoff Leach for triangulation step, an open-source program. The author improved the divide and conquer Guibas-Stolfi algorithm and he got a factor of 4-5 speedup. This is a $O(nlog(n))$ algorithm [13].

The update step move the points to the new position, using the scalar force calculated for each bar. This is the main action taken by the algorithm and this is carried out using a large matrix of movements. Every matrix element ($M_{i,j}$) stores the movement of the i^{th} point which is one end of the k^{th} bar. The total move for n_i node is $P_i = \sum_{j=0}^{j=n-1}(M_{i,j})$. However, only a few bars converge at each mesh point.

This serial version is an easily implemented way to guarantee a correct execution and to facilitate the writing of a correct parallel version.

4. Our GPU-based Algorithm

As argued, Distmesh algorithm is highly suitable for GPUs architectures. Most of the operations, performed by the sequential mesh algorithm, were easily mapped on GPUs multiprocessors. The parallelism in Distmesh code is exploited by dividing the vector operations among the threads. Distmesh loop iterations are distributed to kernel blocks, so each data is fetched by a thread and every thread executes the same kernel.

Figure 1 presents a high-level overview of our parallel GPU-version of Distmesh algorithm. It outlines where the GPU acts as a parallel CPU co-processor in a collaborative way. The initial phase is executed on the CPU. CPU generates the first triangulation and copies points and bars from host (RAM) to device (GPU global memory). CPU launches the GPU kernels function to start the GPU mesh generation process. When GPU concludes, only final positions of points are copied from device to host. Data transference between host and device is performed at initial and final steps of the algorithm. During core operations, data remain at device memory. During kernels execution, bars length and data movements array remain resident in GPU device memory. We implemented our parallel mesh generator in this way, to exploit GPU threads concurrency. In next section, we describe the different steps we designed to run our parallel algorithm of Distmesh on a GPU.

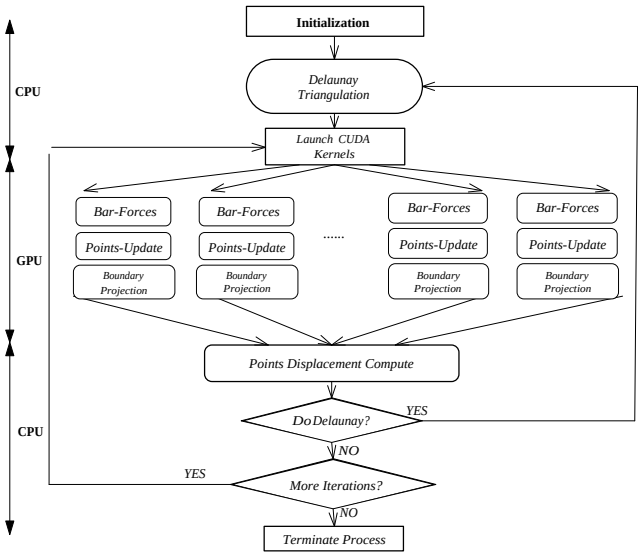


Fig. 1: Mesh Generator Algorithm in CPU-GPU.

4.1 GPU kernels

As a first step in writing the parallel CUDA program, we had to identify those code blocks we could separate from sequential program as a CUDA kernel. Here we explain each kernel written to exploit both data parallelism of the algorithm itself and the GPU architecture. Once written, data on the GPU is persistent unless it is deallocated or overwritten, remaining available for subsequent kernels.

Every node location remains fixed during its total force computing, that's why this task may be parallelized. Figure 2 shows data structures, kernels and the relationship between them. Data structures were selected to avoid the branch instructions in kernels code. This was possible by increasing the compute over the array elements, in order to minimize the use of *if-then-else* control instruction. The general description of each kernel is expressed bellow.

Bar length: We mapped a thread per bar, making every thread compute its corresponding bar length. Threads read every data stored in *Bars* array and store their results in *Length Bar* array. This was a suitable condition for thread computing. Subsequently, we had to do a sum over all elements of the lengths array, and so calculate relax length l_0. This was not a suitable condition for thread computing, we will refer to this later.

Scalar Forces: This kernel launch a thread per length bar. Data are supplied by length array, at global device memory. Every thread applies the same operation to every data, to calculate the resultant scalar force applied to each bar. Then stores the results in a new data structure, *Move*. Move array dimension is in correspondence with the number of bars.

Points Movements: This kernel reads data movement -

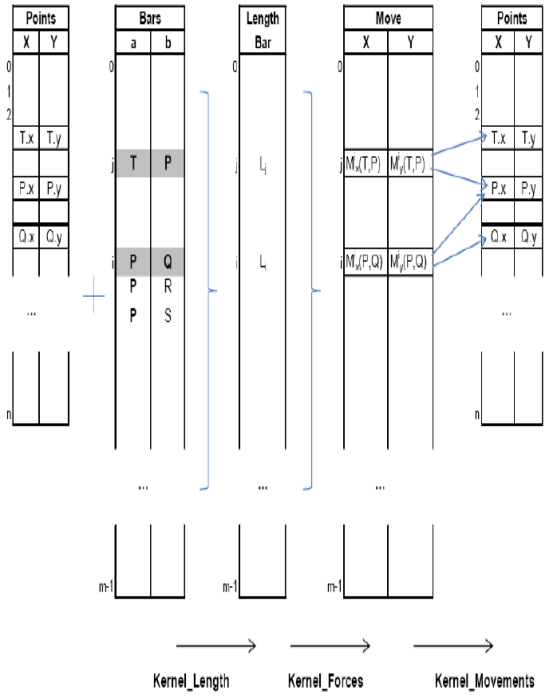


Fig. 2: Data Structures created by the kernels in GPU.

which was calculated with Scalar Forces kernel- in every bar. Each bar contributes to the total movement of all the meshpoints, affecting their extreme points. The final position of each point is obtained by summing all these movements, as shown in Figure 2.

Boundary Projection: Some points go outside the geometry after updating process. This kernel is in charge of projecting the points moved out from the domain, by relocating them to the nearest point in the border. This relocation is performed by calculating the numerical gradient.

The next step in writing a CUDA program, was to manage data transfer between RAM memory and the GPU global memory.

4.2 GPU kernels optimization

As we explained before, the data transfer from host to device is made only twice, before launching the kernels and when GPU process ends. We don't have the bottleneck memory transfer problem during GPU computation [14]. As shown in Figure 3, the data transfer time is 5.7% of GPU time, for the entire process. This graphic was obtained with CUDA Profiler tool. The kernels outlined above are limited by the rate at which the GPU can issue instructions; they are compute bound. To improve the performance, we optimized memory access using shared memory. CUDA uses share

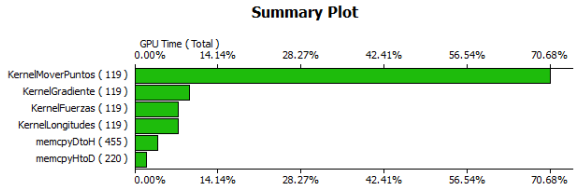


Fig. 3: Kernels and data transference.

memory to help reducing overfetch [15]. To avoid multiple simultaneous accesses to memory, we could efficiently load data arrays from global to share device memory, thus ensuring coalesced readings [16]. CUDA *Atomic Add* functions are the core arithmetic operations to sum array elements and to calculate forces and points movement. These functions ensures that the readings will be done without any interference from other threads. Synchronization is guaranteed by CUDA if multiple threads, in different blocks, access to the same variable to perform *read-modify-write* operations [10][11].

Bars length kernel optimization: In a first version, relax length l_0 was performed by one thread. The other threads remain idle in the meantime. The run time was improved copying lengths bar structure to a new array in share memory and using atomic functions to sum lengths bar. Figure 3 shows its little kernel runtime compared to the total kernels execution time. This kernel uses a CUDA Atomic-Add function on share memory to sum bar lengths.

Points movements kernel optimization: We modified this kernel to avoid using a huge data array. It was possible performing atomic operations. This kernel reads movements stored in move array, as shown in Figure 2, launching a thread per row. Each row represents the scalar force at a bar and it contains x and y component of points movement, then each thread modify the position of two points. This action was optimized by using the CUDA *ATOMIC_FLOAT_ADD* function, obtaining a significant improvement in performance.

We present the experimental results in next section.

5. Experimental Results

In this section we evaluate the computational performance of our GPU parallel version of Distmesh on a platform consisting of a Intel Xeon dual-core processor with 4GB of main memory running at 3.2 GHz, connected to a NVIDIA Tesla C2070 with CUDA driver and runtime version 4.0. This GPU is comprised of 14 streaming multiprocessors (SMs) of 32 streaming processors (SPs) for a total of 448 SPs CUDA cores and its CUDA capability is 2.0.

Table 1 shows CPU and GPU execution time. These measurements of time consider the complete execution of

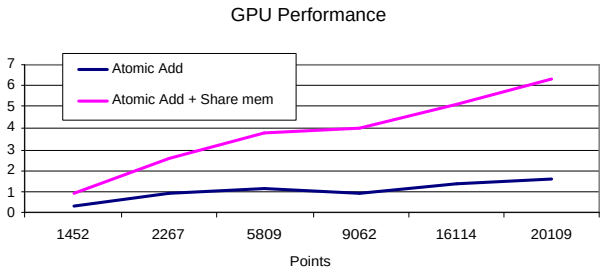


Fig. 4: Comparison of execution time.

Table 1: GPU Version: Execution Time (sg.)

Node Points	*CPU Time*	*GPU Version1*	*GPU Version2*	*Speedup Complete*
1452	0.035	0.130	0.037	0.95
2267	0.290	0.320	0.112	2.59
5809	0.687	0.580	0.183	3.75
9062	1.689	1.820	0.425	3.97
16114	3.388	2.510	0.666	5.09
20109	7.786	4.860	1.239	6.28

the algorithm, including both the executed phase by the CPU and the GPU, including memory transfer overhead. The maximum speedup achieved is: 6.28. GPU execution time was measured performing atomic add operations in *Version 1*, and performing atomic add operations at shared memory in *Version2*. We obtain a significant improvement in the second case. The execution time evolution, for a complete run, is presented in Figure 4.

To ensure consistent results when computing our sequential(CPU) and parallel(CPU+GPU) algorithms, we compared the number of iterations done and how many of those invoked to Delaunay. In Table 2, we present the relationship between Delaunay and Persson iteration, with the aim of showing consistency between CPU and GPU implementation. Rate values are close enough in both cases, allowing ensure satisfactory results, in accordance with original Distmesh results in Matlab. Moreover, we achieve a substantial improvement by measuring only core operations in the GPU version of mesh generator. The speedup reached 53X. We call core operations to GPU kernels outlined in section 4.

6. Related works

In this section we give a brief overview on mesh generator algorithms on GPU. Démian Nave et al. present their approach to parallelizing the Delaunay mesh generation, which can be parallelized in a natural way. This is a similar

Table 2: GPU-CPU execution: Delaunay and Persson iterations.

Node	*CPU*		*GPU*		*Speedup*
Points	*Delaunay*	*Persson*	*Delaunay*	*Persson*	*Kernels*
1261	25	265	23	253	4.77
2827	39	365	30	313	6.81
5025	45	381	35	381	4.94
7851	49	487	38	487	5.09
11313	65	658	44	616	12.08
15395	150	1056	134	983	12.10
20109	231	2083	212	2405	17.35
31409	728	7764	719	6980	53.38

situation to our work. They emphasize the importance of mesh generation algorithm and of Delaunay method in particular [17]. In 2008, Rong et al. present their approach to GPU computing. They enhance Delaunay triangulation using GPUs as a parallel co-processor in charge of the triangulation on a given set points in 2D. The best results are obtained for a large number of points, when they achieve a 53% improvement compared to *Triangle* Delaunay algorithm [18]. An interesting GPU mesh generator algorithm is presented in [19]. The authors propose a two-phase iterative GPU based method, that transforms any 2D planar triangulations and 3D triangular surface meshes into their respective Delaunay form. They used this algorithm to simulate sten deformation, where the geometry of triangulation changes dynamically and requires restore Delaunay conditions to interactive real time levels. This situation is similar to points retriangulation needed in our work, where we use Delaunay triangulation too. We are working on Delaunay parallelization to improve our work, and we are interested in the previous papers.

7. Conclusions

We wish to highlight the GPUs technology suitability to improve performance of mesh generators algorithms. We showed how the efficient Matlab Distmesh algorithm can be parallelized by processing its mesh nodes concurrently and taking advantage of its data structures. Our results gives us an idea of the computing power offered by GPUs and the virtual machine defined by CUDA, which exhibit scalability to programmers. We initially ran our application in a NVIDIA G80 series card; despite being old devices, we obtained good results. Then, we could run the CUDA program in a TESLA card making minimal changes to the kernels code. This architecture enabled us to use Atomic functions in floating point.

We presented in this paper a developmental stage of our work and it shows our initial experiences, which resulted in a significant decrease of algorithm execution time. The Persson method generates high-quality meshes, which were perfectly reproduced for domains in 2D with our parallel

algorithm. This approach requires improving the management of large amount of data when dealing with complex geometries and non-uniform meshes. Anyway, we intended to provide a contribution to this topic development, in the search of high performance in GPUs computing.

8. Acknowledgments

The authors thank to Gabriel Acosta for his help in making this work possible. The work was supported by the research project PICT2007-910-CONICET-ARGENTINA.

References

[1] N. Corporation, *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 2.3*, 2009. [Online]. Available: http://developer.nvidia.com/nvidia-gpu-programming-guide

[2] "Nvidia corporation," LO-tesla-brochure-12-lr.pdf, 2010. [Online]. Available: http://www.nvidia.es/object/LO-tesla-brochure-12-lr.html

[3] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, Mayo 2008.

[4] J. Nickolls and W. Dally, "The gpu computing era," *IEEE Micro*, pp. 56–69, Marzo 2010. [Online]. Available: http://www.computer.org/portal/web/guest/home

[5] U. B. Per-Olof Persson Department of Mathematics, "Distmesh - a simple mesh generator in matlab." [Online]. Available: http://persson.berkeley.edu/distmesh/

[6] J. R. Shewchuk, "Mesh generation for domains with small angles," in *SCG '00: Proceedings of the sixteenth annual symposium on Computational geometry*. New York, NY, USA: ACM, 2000, pp. 1–10.

[7] P.-O. Persson and G. Strang, "A simple mesh generator in matlab." *SIAM Review*, vol. 46, pp. 329–345, 2004. [Online]. Available: http://persson.berkeley.edu/publications.html

[8] T. R. Halfhill, "Parallel processing with cuda. nvidia's high-performance computing platform uses massive multithreading." *Microprocessor Report*, January 2008.

[9] *Optimization. NVIDIA CUDA C Programming. Best Practice Guides.*, NVIDIA Corporation, Cuda Toolkit, July 2009.

[10] J. Nickolls, I. Buck, M. Garland, and K. Skadron, *Scalable Parallel Programming*. ACM QUEUE, April 2008.

[11] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, E. Inc., Ed. Morgan Kaufmann, 2010.

[12] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, 2008.

[13] G. Leach, "Improving worst-case optimal delaunay triangulation algorithms." in *In 4th Canadian Conference on Computational Geometry*, 1992, p. 15.

[14] J. Stratton, S. Stone, and W.-m. Hwu, "Mcuda: An efficient implementation of cuda kernels on multi-cores." University of Illinois at urbana-Champaign - Center of Reliable and High-Performance Computing., Tech. Rep., April 2008.

[15] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, and S. S. Stone, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda abstract," in *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. ACM*, 2008, pp. 73–82.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1370–1380, 2008.

[17] D. Nave, N. Chrisochoides, and L. P. Chew, "Guaranteed-quality parallel delaunay refinement for restricted polyhedral domains," in *Computational Geometry (COMGEO)*, ser. 28, 2004, pp. 191–215.

[18] G. Rong, T.-S. Tan, T.-T. Cao, and Stephanus, "Computing two-dimensional delaunay triangulation using graphics hardware," in *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*. New York, NY, USA: ACM, 2008, pp. 89–97.

[19] C. Navarro, N. Hitschfeld, and E. Scheihing, "A parallel gpu-based algorithm for delaunay edge-flips," in *27th European Workshop on Computational Geometry (EuroCG)*, I. M. Hoffmann, Ed., 2011, pp. 75–78.

GPU-Based Implementation of JPEG2000 Encoder

M. Ahmadvand[1], **and A. Ezhdehakosh**[2]
[1] Department of Computer Engineering and IT, Hamedan University of Technology, Hamedan, Iran
[2] Department of Computer Engineering and IT, Amirkabir University of Technology, Tehran, Iran

Abstract - *JPEG2000 has become one of the most rewarding image coding standards. It provides a practical set of features which weren't necessarily available in the previous standards. The features were realized as a result of two new techniques, namely the Discrete Wavelet Transform (DWT), and Embedded Block Coding with Optimized Truncation (EBCOT). The complexity of EBCOT Tier-1 makes its implementations very difficult and time consuming.*

In this paper, we focus on accelerating JPEG2000 encoder by using general-purpose processing on Graphical Processing Unit (GPU). We use CUDA platform to implement DWT and EBCOT Tier-1 as the most important sections of JPEG2000. Resulting implementation of proposed architecture performs very well compared to other available implementations.

Keywords: *JPEG2000, GPU Computing, CUDA*

1 Introduction

JPEG2000 has become one of the most rewarding image coding standards. It provides a practical set of features which weren't necessarily available in the previous standards. JPEG2000 [1][2] offers numerous advantages over JPEG. These advantages include: ROI (Region Of Interest) coding, quality vs. resolution compression, lossless and lossy compression, progressive image compression/transmission by resolution/quality, random code-stream access and error resilience. Such characteristics add to the functionality of a system that is employing JPEG2000 as an image compression technique. The features and performance of JPEG2000 make this standard superior to JPEG. The features were realized as a result of two new techniques, namely the Embedded Block Coding with Optimized Truncation (EBCOT) [3]-[6] and Discrete Wavelet Transform (DWT) [7]. The complexity of EBCOT Tier-1 makes its implementations very difficult and time consuming.

During the process of encoding, an image is partitioned into data matrices called Tile-components. Each Tile-1 component is then coded separately. The process of coding is made up of different sections. These sections are depicted in Figure 1 and each is described below.

1.1 Component Transform

This section is optional in JPEG2000 and is used to improve compression efficiency [3]. The transform converts the RGB data into another color representation, with a luminance (or intensity) channel and two color difference channels.

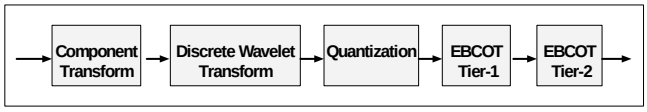


Figure 1. JPEG2000 encoder block diagram

1.2 Discrete Wavelet Transform (DWT)

DWT [7] is a domain transform that transforms an image Tile-component from special domain to frequency domain and provides a special decorrelation. This transform can be executed for as many levels as necessary. The spatial decorrelation provided by the DWT improves as the number of transform levels increases. The output of each level of DWT is categorized into four sub-bands. DWT can be performed either by the traditional convolution based filter or by the lifting scheme based filter which has lower computational complexity compared to the former filter.

1.3 Quantization

Quantization [4] is the process by which the sub-band samples generated by the DWT are mapped onto quantization indices for coding.

1.4 EBCOT Tier-1

This section receives the quantized wavelet coefficients and encodes them into bit-streams. These coefficients are sliced into code-blocks before they are fed into the EBCOT Tier-1 [4]. EBCOT Tier-1 is composed of two parts: Bit-Modeler and MQ-Coder [5][6]. Bit-Modeler is a bit-plane (a matrix that contains all the bits of the same order of all the coefficients of a code block) coder. A Bit-Modeler exploits the symmetries and redundancies within and across the bit-planes and generates corresponding contexts for each bit. After the context is generated, the MQ-Coder will code the bits (decisions) based on their associated contexts.

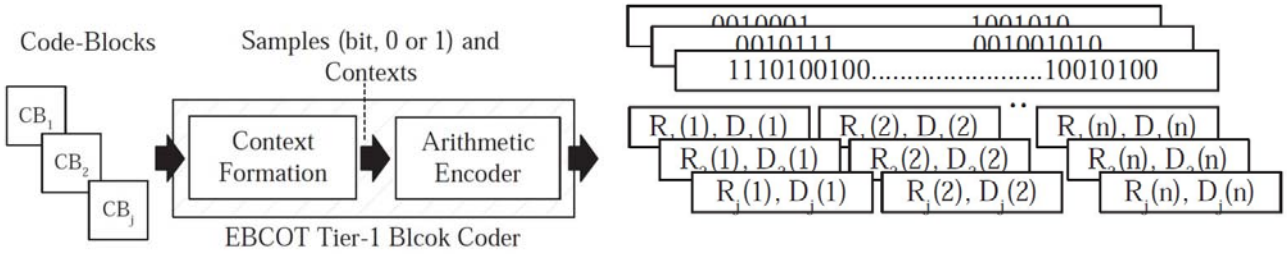


Figure 2. Entire coding process of EBCOT Tier 1 block coder

Table 1 Run time percentage for different modules in JPEG2000 encoder

Operation	Lossy	Lossless
Component Transform	10.1	3.64
DWT	25.14	10.41
Quantization	6.4	N.A.
EBCOT Tier-1	44.86	67.35
EBCOT Tier-2	13.5	18.6

1.5 EBCOT Tier-2

EBCOT Tier-2 [2] is for rate allocation. The rate allocation is responsible for acquiring the highest quality for the output while maintaining a predetermined resolution, or acquiring the highest resolution while maintaining a predetermined image quality.

The execution time of different modules in the JPEG2000 algorithm is presented in Table I. It is noted from this table that DWT and EBCOT Tier-1 algorithms, as the main modules in JPEG2000 standard, occupies %70 of the execution time of the whole procedure. In this paper, we focus on accelerating JPEG2000 encoder by using general-purpose processing on Graphical Processing Unit (GPU) [9]. We used CUDA [10] platform to implement DWT and EBCOT Tier-1 as the most important sections of JPEG2000. Resulting implementation of proposed architecture performs very well compared to other available implementations.

This paper is organized as follows: in the next section a deep analysis of the EBCOT Tier-1 will be presented. In section III DWT transform will be described. In the next section GPU computing using CUDA will be explained. Our proposed implementation is introduced in sections V-VI. Experimental results are presented in section VII followed by the conclusion.

2 EBCOT Algorithm

EBCOT [3] is a two-tiered coder, where the first tier is a block coder and the second tier is for rate-distortion optimization and bitstream formation. Although Tier 2 is part of EBCOT algorithm, the practical implementation detail is not defined in the standard and not restricted in the design of an encoder.

Tier 1 of EBCOT is actually a context-based adaptive arithmetic encoder. Code-blocks are independently coded by this block coder into sub-bitstreams. According to the functionality, the block coder can be further partitioned into two steps, Context Formation (CF) and Arithmetic Encoder (AE), as shown in Figure 2. The transformed coefficients of a code-block are coded bit-plane by bit-plane from most significant bit-plane (MSB) to less significant bit-plane (LSB) instead of coefficient by coefficient. CF scans all bits within a bit-plane of a code-block in a specific order, and generates corresponding contexts for each bit by checking the status of the neighborhood bits. AE then encodes each bit according to the adaptively estimated probabilities from its contexts. The outputs of the arithmetic encoder are the sub-bitstreams of each compressed code-block data. Also, the rate and distortion information are calculated for each pass for subsequent tier 2 processing.

Before CF, the quantized (in lossy mode) or non-quantized (in lossless mode) wavelet coefficients are converted from two's complement representation to sign-magnitude format. The scanning order in a code-block is from the most significant bit-plane of the magnitude part to the least significant bit-plane. A sample is called "significant" after the first "1" bit is met while encoding the magnitude part from the MSB to the LSB, and it is called "insignificant" before the first "1" bit appears. The sign bit, is coded immediately after the first "significant" bit is coded. Within a bit-plane, every four rows form a "stripe," and the scanning order is stripe by stripe from top to bottom. In every stripe, data are scanned bit by bit from top to bottom, and column by column from left to right. To improve embedding, fractional bit-plane coding method is used. Embedded coding, which is useful for scalability and for efficient rate control, is actually one of the main features of JPEG 2000. Under this fractional coding method, one bitplane is further decomposed into three passes according to coefficients' significant situations. While scanning from the top bitplane, all-zero bit-planes are skipped. When the first nonzero bit-plane is found, only pass 3 coding is used to encode all bits in this bit-plane since no bits will be coded in pass 1 and pass 2 according to the coding rule. The subsequent bit-planes are scanned three times each following the scanning order described above. The first scanning is for pass 1, and is followed by pass 2 and pass 3 scanning. Each bit in a bit-plane is encoded in one of the three

passes. Pass 1 is named "Significant Propagation Pass." During pass 1 scanning, those samples that are currently insignificant, but have at least one immediate significant neighbor are coded first. Clearly, these samples are most likely to become significant. Pass 2 is called "Magnitude-Refinement Pass." Samples that have become significant in previous bit-planes are coded in this pass. The last pass, pass 3, is "Clean-Up Pass." Samples not coded in the first two passes are coded in this pass. The two bits that are coded in pass 2 in this bit-plane have become significant in the previous bit-plane. Those bits coded in pass 1 are near those two pass 2 positions due to the significance propagation characteristic.

Once a bit is checked and decided to be coded in one pass, its context is generated according to the status of its neighbors using four coding primitives, Zero Coding (ZC), Run-Length Coding (RLC), Sign Coding (SC), and Magnitude Refinement (MR) primitives. There are total 19 contexts defined in JPEG 2000 [3]. The context of a bit should be generated and sent to arithmetic encoder along with the bit to be coded. Among the four coding primitives, ZC and SC primitives are used in the first and the third pass, MR primitive is used in the second pass only, and RLC primitive is used in the third pass. ZC and MR primitives check the eight immediate neighbors' significant states to decide the context, and the RLC primitive is applied when the four bits in a column do not have any significant neighbors. The SC primitive has to check the four immediate neighbors' sign and significant states.

After the context is generated, the arithmetic encoder will code the bits (decisions) based on their associated contexts, called MQ-Coder. Generally, a BAC encodes a code-stream consisting of a sequence of symbols. Each symbol (logic '0' or logic '1') is classified into one of these categories: the More Probable Symbols (MPS), and the Less Probable Symbols (LPS), based on the probability of their occurrence. In BAC an interval is considered as a probability model. This interval is divided into two subintervals, each one, corresponding to the probability of each symbol. When a symbol occurs, the subinterval associated with that symbol becomes the new interval. The recursive splitting of the current interval continues until all symbols are received. When the last symbol is received the characteristics of the last subdivided interval represents the encoded data.

As indicated above, BAC algorithm requires many multiplication operations in order to encode each symbol, and multiplication is time consuming. In addition, since a compressed data will only be generated when the last symbol of an input stream has been received by the encoder, serious loss of data occurs when the last symbol of a stream is not received. Finally after each subdivision of the probability model, the precision required for presenting the new interval increases. This leads to an increase of the required storage space for the interval values.

The MQ-Coder is an adaptive BAC implementation used in the JPEG2000 standard. MQ-Coder has eliminated multiplications by choosing special extremes for the intervals that are used in the probability model. In addition, the MQ-Coder periodically sends out the last byte of the stream which represents the encoded data, therefore addressing the problems associated with the increasing precision and compressed data being generated only after receiving the last symbol.

3 Discrete Wavelet Transform (DWT)

Discrete Wavelet Transform (DWT) [7] is a broadly used digital signal processing technique with application in diverse areas. DWT allows us to study a digital signal in different resolutions as sets of coarse and fine values. Wavelet transforms are used in domains of digital speech recognition, multi-resolution video processing or data compression. In the context of JPEG2000 standard, DWT is the key prerequisite of the compression process. Most of advanced features of JPEG2000 rely on DWT as well as the superior low-bitrate performance does.

JPEG2000 standard species use of LeGall (CDF) 5/3 DWT filter-banks [8] for lossless compression process and Daubechies-Feauveau (CDF) 9/7 DWT filter-banks [8] for lossy processing. Wavelet transforms can be implemented by convolution or by lifting scheme.

The advantage of lifting scheme over convolution is in reduced memory and computational complexity. Lifting scheme allows for in-place data manipulation and reduces memory dependencies.

Lifting scheme analysis [7] proceeds as follows. An input signal is split into even and odd subsequences denoted as and respectively. These values are further modified using alternating prediction (denoted as p) and update (denoted as u) steps. In the prediction step, the algorithm takes an odd sample in a turn and substracts a linear combination of its (even) neighbors from it; a prediction error is formed:

$$d_i^1 = d_i^0 - p(s_i^0 + s_{i+1}^0) \tag{1}$$

In the update step, a linear combination of already modified adjacent odd samples is added to each even sample and updated even sequence is formed:

$$s_i^1 = s_i^0 + u(d_{i-1}^1 + d_i^1) \tag{2}$$

The output of the last update stage, , is actually a low-pass output of DWT filter and similarly output of the last prediction stage, is a high-pass output of the filter. So the result of the wavelet transformation is a signal divided into low-pass and high-pass subbands.

2D signals (e.g., images) are usually transformed in both dimensions. 1D DWT transform is first applied to all rows then to all columns resulting in four subbands LL, HL, LH, and HH. The LL subband is an approximation of the original signal and can be further transformed recursively.

4 GPU computing based on CUDA platform

CUDA [10] is software and hardware platform designed for general purpose computing on GPUs. GPUs have a parallel architecture capable of running thousands of threads in parallel. In CUDA computing model, such threads are grouped into so called thread blocks. Threads within a block can cooperate among themselves by sharing data through a shared memory. As opposite to a large global memory, shared memory is relatively small and very fast. The advantage of global memory is that it is accessible to all threads, whereas shared memory is visible only to threads of the block. Common work flow is to copy data from RAM to global memory of GPU. Once data is ready in global memory, a GPU program can be executed. Each thread block initially fetches a small portion of data from the global memory into the shared memory. Data is then processed by threads in the block and the result is moved back to the global memory.

The global memory access pattern is perhaps the most important performance consideration in programming for the CUDA architecture. In a nutshell, when 16 adjacent threads access adjacent locations in global memory then memory loads and stores are coalesced in one transaction.

5 GPU-Accelerated implementation of DWT

The key part of our GPU-accelerated DWT is the design how to split the work between thread blocks in order to provide maximum utilization of the GPU. Since we need to compute the transform in both dimensions, it is natural to choose a 2D partitioning of source image data. Also size and shape of thread blocks needs to be determined. Because the lifting scheme algorithm alternately works with even and odd samples, an efficient approach is to have one even and one odd data sample per each thread in a block, i.e., to have twice as much data samples as threads in each block. Resulting partition of image data. Each thread block has its dimensions Bx×By where Bx = Dx, By = Dy/2 , and Bx,By and Dx,Dy denote number of threads and samples in horizontal and vertical direction respectively. Thus that thread blocks are of rectangular shape while data blocks are of squared shape.

Each thread has its exact position within the block, which is determined by indices Tx, Ty where $0 \le Tx \le Bx-1$ and $0 \le Ty \le By-1$. We can see that threads overlap only the upper half of the block, i.e., MAX(Ty) = Dy/2-1, which means that threads are directly mapped only to samples in the upper half of block data and we will have to change this mapping to be able to process both halves.

The first step of computation is to fetch image data from global memory into fast shared memory. It is crucial here to comply with coalesced global memory access. Considering the proposed data partitioning, each thread loads corresponding data sample into the upper half first, and then into the lower half of data block. The horizontal block size should be multiple of 16, so that coalesced access is not broken by thread block misalignment.

DWT coefficients are then computed according to lifting scheme relations 1 and 2. To calculate first dimension of the transform, DWT filters are applied to every row separately. Afterwards, each row contains a sequence of interleaved coefficients of low-pass and high-pass subbands. Each particular prediction and actualization step is calculated respectively as follows:

$$s[T_x][2T_y+1] = s[T_x][2T_y+1] + p.(s[T_x][2T_y] + s[T_x][2T_y+2]) \quad (3)$$

$$s[T_x][2T_y] = s[T_x][2T_y] + u.(s[T_x][2T_y-1] + s[2T_y+1]) \quad (4)$$

Where Tx and Ty determine the thread position in horizontal and vertical direction respectively and s[x][y] is the shared memory 2D array. Note that we propose transposed thread mapping for efficient data processing as follows. Threads are directly mapped into the upper half of block only, so that we have to change the thread mapping to be able to process whole block. In (3) and (4), we have swapped 3 thread indices Tx, Ty so that the threads cover the left half of the data block instead of the upper half which was covered originally. (3) then predicts all odd samples and (4) updates all even samples in the block. To calculate the second dimension of the transform, we just apply same filters to the columns.

The result of the application of lifting filters to rows and columns is composed of coefficients of four DWT subbands. Coefficients of LL and HL subbands are alternately located on even rows and LH and HH coefficients on odd rows of the shared memory s.

The final step of the CUDA-based transform is to move result from shared memory back to global. Particular subbands, however, needs to be stored separately in global memory. Because there are twice as much data samples as threads in the block, we store even lines first. Even lines contain all LL and HL samples and because those are interleaved, we use first half of threads to store all LL samples and second half to store HL samples. The access to the global memory is hereby coalesced.

Note that proposed implementation of DWT is optimized for maximum performance and its limitation is that it does not take into account sample values exchange between blocks borders. The proposed algorithm does not introduce any visual artifacts provided both forward and reverse transformations work with the same data blocks dimensions.

6 GPU-Accelerated implementation of EBCOT Tier-1

Architecturally, the parallel algorithm for the bit plane coder is optimized to match the existing graphics hardware. A number of code blocks are processed independently and each code block samples should be processed in parallel using the

state prediction method. Hence, each work group is in charge of handling one code block, and multiple processing elements (PE) of the work group can process the samples in parallel. In particular, the usage of memory resources has to be carefully optimized, control flow operations must be minimized as they result in costly processor stalls, and the workload must be distributed so as to maximize hardware resources occupancy and hide memory latency when stalls are unavoidable.

In GPGPUs any flow control instruction (i.e. if, switch...) can significantly affect the instruction throughput by causing threads of the same parallel thread block to diverge; that is, to follow different execution paths. GPGPUs provide great arithmetic capability at low hardware cost, but to achieve this goal, the cores in each multiprocessor often share only one instruction decoder and one small branch control unit. Therefore a single instruction is executed over N threads in parallel, where N is specific to the hardware chip and often has value of 32 or 64. As a result, control instructions and branch divergences on GPGPUs tend to be very expensive.

Unfortunately, the context formation process in JPEG2000 requires many control operations. For example, when the BPC scans a 3×3 window of neighboring samples (8 neighbors of a given sample), the algorithm may take 256 different execution paths. Additionally, the BPC needs to select one out of 19 contexts based on that information. If the bitplane coder (BPC) is implemented with a standard switch/case construct its performance would be bound to be very low. Fortunately, the context decision rules are predefined so look-up-tables (LUT) for context formation can be constructed to avoid the branching control flow. A LUT should have 256 entries, where the indices for the entries are formed from the 8 neighbor state bits and the value is selected based on the context rule. However it is inefficient to concatenate the state bits to form a LUT index every time. Therefore the state bits are stored in a 16-bit state flag instead, as shown in Figure 3. Each sample has one corresponding state flag that stores the state information itself and the state bits of its neighbors. This organization allows the BPC to easily retrieve an LUT index by applying a bit mask.

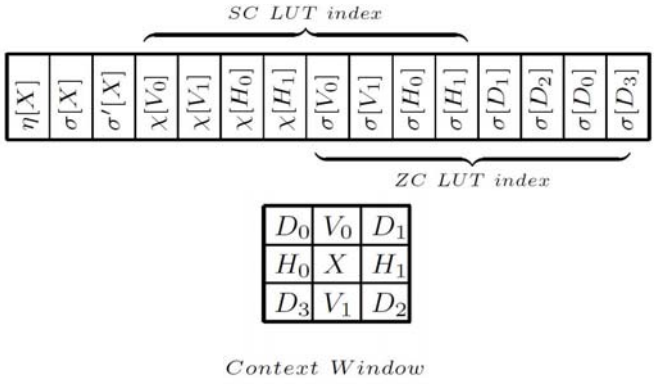


Figure 3. 16-bit state flag of a sample. The index of ZC or SC LUT can be extracted from the flag by applying a binary mask

It is very critical to optimize memory usage for the BPC where the context formation process executes massive numbers of memory and arithmetic operations. Particularly on GPGPUs, an efficient memory utilization can not only significantly reduce the latency but it also can increase the resource occupancy of computational resource to speedup the computing time.

The first optimization considered is to efficiently allocate different sets of data into the most suitable of memory blocks based on the application's demand to reduce latency and conflict. When the BPC processes one sample, it not only refers to the sample but also to its 8 neighbors. Consequently, there is a high degree of memory conflicts in the BPC, particularly in parallel BPC where multiple threads concurrently access different samples. However, both the memory conflict rate and memory latency can be dramatically reduced with very fast, multi-way shared memory that resides locally on chip. The shared memory on modern graphics cards has from 16 to 32 banks which can be accessed independently with a latency of only one clock cycle. It is also very important to optimize allocation of the context output buffer since it is the intermediate buffer for context formation and arithmetic coder. Storing this buffer in the global memory would be very inefficient since the BPC would write to global memory and then the arithmetic coder would have to read back from global memory immediately after BPC write. Therefore this output buffer should be also stored in on-chip shared memory. Additionally, since the BPC refers to the LUTs and the state flags very frequently, these data structures should also be placed in the shared memory as well. The LUTs are read-only and small enough to reside in fast the constant cache memory. The code blocks are initially stored in the off-chip global memory then each multiprocessor will copy its respective code block into its shared memory. The LUTs are stored in constant memory and fetched to multiprocessors' constant cache at runtime.

After the data sets are efficiently allocated into selected memory regions, the utilization of memory, especially shared memory, should be minimized to increase the multiprocessor occupancy of the GPGPUs.

The multiprocessor occupancy is defined as the ratio of the number of resident warps to the maximum number of warps supported on a multiprocessor of a GPU. Typically the higher occupancy the better multiprocessor can hide the warps latency and increase ALU utilization which will yields better speedup. Each multiprocessor on a GPU has a set of registers and a small amount of on-chip shared memory. These resources are shared among the active thread warps. Therefore the lower shared resources are utilized by a particular warp, the higher number of warps can reside in a multiprocessor. The compiler can attempt to minimize register usage but the utilization of shared memory must be optimized by the programmer. Additionally, the number of threads per work group should be large enough, at least 4× of the warp size, to achieve the best performance.

However, it is not simple to reduce shared memory utilization in the parallel EBCOT Tier-1 coder since it

depends on the code block size which is one of the key parameters that determine JPEG2000 compression efficiency. The code block size is often varied from 8×8 to 64×64 samples, but using the largest allowed configuration is preferred because the larger code block size the better compression efficiency. On the other hand, a code block size requires a large shared memory buffer which may reduce the multiprocessor occupancy and hence reduce the performance speedup. In a naive implementation, at least 24 Kbytes of shared memory are needed to store a 64 × 64 code block and the two respective state flag, while a 8 × 8 code block requires only a small 256-byte buffer. The results show that by changing code block size from 64 × 64 to 8 × 8, the compression efficiency drops about 25% on image Bike. But by decreasing code block size the multiprocessor occupancy can be significantly improved from 17% to 50%. There is an unusual result with the 8×8 code block where the occupancy no longer increases.

It clearly shows that the large code block significantly decreases the multiprocessor occupancy which results in significant speedup drop. To overcome this problem, previous studies had to compromise compression efficiency to use a GPU-affordable code block size such as 8 × 8 but this is an impractical size. This study therefore manages to design a special strategy that can handle large code blocks with low utilization of shared memory.

7 Experimental results

This section presents the result of our proposed implementation. The parallel JPEG2000 coder is implemented using CUDA on the Nvidia SDK 3.2 running on an Nvidia GTX480 graphic card. The reference CPU platform uses an Intel Core i7, with 12GB RAM running at 2.8GHz.

There are several popular versions of JPEG2000 compression software running on CPUs, including JasPer [11], and OpenJPEG [12]. JasPer is chosen to compare against the GPU implementation since it is an open source program, with fully accessible source code, and very good performance. The image test set includes the most popular JPEG2000 test images (bike, woman, cafe, and lena).

Table II compares runtime for JasPer and the GPU implementation JPEG2000 coder. The GPU-based JPEG2000 encoder implementations are more than 17× faster than the JasPer implementation.

Table 2. Runtime comparison of our method with JasPer

Image	Time (ms)		Speedup
	Our GPU	JasPer	
lena	15.23	238.74	15.7
cafe	267.36	5214	19.5
bike	284	4398.28	15.49

8 Conclusion

In this paper, the design and development of a novel PEG2000 encoder are presented. The parallel algorithm can process data at the sample-level. In particular, this paper is the first to presents a fully parallel solution for the arithmetic coder and DWT in JPEG2000. The implementation of the JPEG2000 coder leverages widely available and massively parallel GPGPU hardware and provides a 17× performance speedup compared to the JasPer software implementation. It is believed that even greater speedup is possible with full 64-bit hardware support. Additionally, the proposed parallel algorithms are potentially applicable to a wide range of image Processing and data compression applications.

For future work, the emphasis will be on further improved implementations of the arithmetic coder. In addition, the Tier-2 routines can be parallelized in order to have a complete JPEG2000 encoding flow running on a GPU platform. Another research direction is that of implementing the proposed parallel solutions on different parallel hardware platforms to compare the different architectures on the performance and optimization strategies.

9 References

[1] D. S. Taubman and M. W. Marcellin, JPEG2000 Image Compression Fundamentals, Standards and Practice. Kluwer Academic, 2002.

[2] ITU-T, "T.800 : Information technology-JPEG 2000 image coding SYSTEM: CORE CODING SYSTEM," 2001.

[3] D. Taubman, "High performance scalable image compression with EBCOT," Image Processing, IEEE Transactions on, vol. 9, no. 7, pp. 1158 –1170, jul. 2000.

[4] K. Varma, H. Damecharla, A. Bell, J. Carletta, and G. Back, "A fast JPEG2000 encoder that preserves coding efficiency: The split arithmetic encoder," Circuits and Systems I: Regular Papers, IEEE Transactions, vol. 55, no. 11, pp. 3711 –3722, dec. 2008.

[5] M. Ahmadvand, O. Fatemi, H. Badakhshannoory, M.R. Hashemi, "A Novel Pipelined Architecture for JPEG2000 MQ-Coder with Reduced Hardware Resource Requirement", 24th Picture Coding Symposium, December 2004.

[6] M. Ahmadvand, A. Shahrokhi, O. Fatemi, "A High-Speed Pipelined Architecture for MQ-Coder of JPEG2000 Standard", 22nd Queen's Biennial Symposium on communications, (2004).

[7] J. Matela, "GPU-Based DWT Acceleration for JPEG2000," in Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, Brno : NOVPRESS s.r.o., 2009, pp. 136-143.

[8] [1] Tenllado, C., Setoain, J., Prieto, M., Pinuel, L., Tirado, F.: Parallel implementation of the 2d discrete wavelet transform on graphics processing units: Filter bank versus lifting. Parallel and Distributed Systems, IEEE Transactions on 19(3) (2008) 299-310.

[9] Nvidia Coporation, OpenCL Programming Guide for the CUDA Architecture, Version 3.2. CUDA SDK 3.2, 2010.

[10]NVIDIA: NVIDIA CUDA Programming Guide 2.0. (2008)

[11]M. Adam, "Jasper JPEG 2000 compression software," Available at http://www.ece.uvic.ca/mdadams/jasper/

[12]OpenJPEG, "OpenJPEG JPEG 2000 compression library," Available at http://www.openjpeg.org/

Solving Sudoku using Particle Swarm Optimization on CUDA

Jason Monk, Kevin Hanselman, Robert King, Raymond Flagg
Yifeng Zhu PhD., Bruce Segee PhD.
Department of Electrical and Computer Engineering
101 Barrows Hall
University of Maine
Orono, ME, 04469

***Abstract*—Sudoku is a popular puzzle utilizing 81 squares in a 9x9 grid consisting of nine 3x3 boxes. The digits 1-9 can each appear only once in a given row, column, or box. This paper describes the implementation of Particle Swarm Optimization (PSO) to solve sudoku puzzles using GPU processing. This PSO uses our open-source PSO framework that takes advantage of CUDA-enabled GPUs. Although each row contains nine digits, permutations of nine digits can be represented as eight "picks". To find a solution each of the nine rows was treated as a permutation. This reduced the problem dimensionality from 81 to 72. With suitable parameters the algorithm was able to solve multiple sudoku puzzles. This paper describes the implementation of the algorithm , the fitness function used, and the effects of variation on PSO parameters. The original PSO framework and the Sudoku code described in this paper are available online.**

I. Introduction

An open source framework for implementing Multi-swarm Particle Swarm Optimization on GPUs using CUDA was developed and discussed in [1] . The framework was previously demonstrated on a problem to optimize the parameters of a PID controller. This was a relatively well behaved problem in a three dimensional space. It was shown that by utilizing GPU processing, the problem could be solved many times faster than was possible using the CPU.

In the previous paper it was claimed that the framework could easily be extended to other problems and higher dimensional spaces. In this paper we utilize the framework to find solutions to Sudoku puzzles. These represent a class of problems in a very high dimensional space . Furthermore, the space has large number of local minima that occur at large distances from the global minimum.

We believe that the solution of Sudoku puzzles represents a very significant optimization problem. In this paper we show that 1) the open source Multi-Swarm Particle Swarm Optimization Framework that we have previously developed is in deed general enough to extend to this problem, 2) that the dimensionality of the problem can be reduced using permutations, 3) that the use of GPUs is crucial for reducing run times from multiple days to hours, 4) that PSO can indeed solve Sudoku puzzles, and 5) that the choice of PSO parameters has a huge impact on the time to find a solution.

II. Sudoku

Sudoku is a logic puzzle that has been extremely popular in the US since about 2005 and is based around using rules and logic to determine where numbers belong. This section describes basics of the sudoku puzzle as well as its origination.

A. Origin

Sudoku is often thought to be of Japanese origin, but this is not actually true. [2] Modern Sudoku first appeared in American papers in the 1980s, but did not become popular in the US until 2005. Although the puzzle we know today only dates back around 30 years, similar puzzles originated in the late 19th century in France, where puzzles with very similar solutions to sudoku originated. [3]

B. Rules

The rules to a sudoku are to fill a nine by nine grid with nine three by three sub-boxes such that only one instance of each of the digits one through nine is contained in each column, row or box. This means in any given column or row there should no repeated numbers and no numbers other than one through nine.

C. Related Work

This was not the first time that Sudoku had collided with biologically inspired algorithms. Sudoku puzzles have been solved by both GA and GPSO in the past. This, however, is the first time using a more generic PSO and adjusting the fitness function accordingly.

Sudoku puzzles were solved by Mantere and Koljonen 2007 [2]. In the same year, Sudoku puzzles were also solved by Moraglio et al. [4], where they introduced combinatorial-based PSO algorithms (GPSO), these spaces were similar to the pick space used in this paper. Sudoku puzzles were confirmed to be solvable using GPSO by Jilg and Carter 2009 [5].

III. Particle Swarm Optimization

PSO was originally developed by Eberhart and Kennedy [6] in 1995. It is heavily biologically inspired and it mimics behaviors that can be seen often in various types of flocking or swarming animals. It is not a complex algorithm and performs very well in continuous spaces that do not have analytical solutions. PSO is an iterative algorithm in that it moves each particle, calculates fitness of its location and then repeats the process.

A. Inspiration

Particle Swarm Optimization was inspired by animals in nature. Many different types of animals travel in groups, and often they benefit from the knowledge of each other. In a school of fish, each individual can benefit from the others knowledge regarding food and predators. Flocks of birds can cover larger areas by spreading out, and informing the flock of any food found. PSO mimics this in that each particle has knowledge about the fitness (or happiness) of itself and of other particles in the swarm, and tends to move toward regions of better fitness.

B. Algorithm

As mentioned previously, particle swarm optimization is based on having many particles, or candidate solutions, moving around an error/solution space. This implementation considers the best particle the one with the lowest fitness value; however it would work just as well looking for particles with larger fitness values.

Each of the particles tries to move to a solution that is better than its current one. To do this, a particle moves in the direction of the best location of the swarm and the best location it has found so far. The particle does a random weighting of each so it will be heading towards both to some degree. The following is the equation that describes the movement towards the particle's best location as well as the swarm's best particle's location.

$$mov = rand()p_{weight}(p_{best}p) + rand()s_{weight}(s_{best}p) \quad (1)$$

Where mov is a vector in parameter space representing a movement, p_{weight} and s_{weight} are scalars chosen by the programmer, p_{best} is a position in parameter space representing the location where a particle has been the happiest, s_{best} is a position in parameter space representing the location of the happiest particle in the swarm, and p is the current particle location. $rand()$ is a randomly selected floating point number between 0 and 1.

Since mov is a vector in parameter space it generalizes to any number of dimensions depending on the parameter space. In the case of no momentum this vector is calculated and then added to the location each iteration.

The algorithm often performs better when each particle's velocity has some momentum associated with it. [7] In the framework used, the velocity was implemented with momentum, such that the new movement was composed of 90% of its old movement plus the new movement calculated above. This helps prevent particles from getting stuck in any single location, such as local minima. A velocity of the particle is calculated and saved each iteration using the following equation.

$$vel_{new} = .9vel_{previous} + mov \quad (2)$$

Where vel_{new} is a vector representing the velocity or the amount that the particle will move this iteration, $vel_{previous}$ is the velocity from the previous iteration, and mov is a vector calculated above in equation 1. Once the velocity is calculated it is added to the particles location every iteration.

C. Related Work

Our PSO framework was not the first one to reach the GPU scene; however it was the first with the goal of making the PSO framework more accessible. Zhou and Tan 2010 [8] implemented a PSO algorithm using GPUs. Rather than using multiple swarms it used a triggered mutation system on top of a standard particle swarm optimization. They were able to achieve a speedup of 25x with this system.

An asynchronous implementation of PSO was created by Mussi et al. 2011 [9]. This allowed each particle to run iterations at its own rate (which was very fast). However this was limited by the fact that only one particle was allowed per block, and the maximum number of blocks that can run in parallel limits the swarm size.

Vanneschi et al. 2010 [10] tested a multi-swarm system where the best particles from one swarm were passed to the next swarm to replace the worst particles, this was done in a ring setup of several swarms. They exchanged this information every 10 steps. This was implemented again by Solomon et al. 2011 [11].

IV. Parallel Particle Swarm Optimization Framework

The Parallel Particle Swarm Optimization Framework was an implementation of the Particle Swarm Optimization algorithm that would use CUDA and be flexible to a number of different problems. The framework was designed so that a programmer with knowledge of C, but minimal knowledge of CUDA could modify the program to solve a large range of problems. In this case the framework was modified to solve Sudoku puzzles.

A. Multi-Swarm

Since each swarm is at some point in time entirely held in CUDA shared memory, the size of each swarm is limited. The following equation defines how many particles can be in a single swarm based on the dimensionality of the problem. [1]

$$Max\#ofPart = \frac{16,384}{8 + 12 * DIM} \quad (3)$$

Since in our problem each particle will be moving within a 72 dimensional space, 72 can be substituted, giving the actual maximum number of particles.

$$Max\#ofPart = 18 \tag{4}$$

Because of this limitation the framework allows for multiple swarms to be run in parallel. To allow the swarms to cooperate in some way, particles are swapped every 1000 steps. Each time a swap occurs there is a 1% chance that any 2 particles will be swapped.

B. Modifications to Code

The only programming required by the framework are generally modifying PSO parameters, such as weights, as well as rewriting the fitness function. The first problem tested was tuning of PID Controller parameters. While this problem was good for its computational intensity, it had a very small memory footprint. When using this framework to implement a sudoku solver a bit more memory was required.

The added dimensionality of this problem will still be held within the particles individual location; however the information specifying constraints to the puzzle is common to all particles. To make this quickly accessible to all threads it was stored in constant memory on the GPU. This memory is cached for all threads to make it easily accessible. Both the "pick" space and the solution space constraints were stored in this constant memory, which will be described in section IV-G.

C. Sudoku Fitness Development

For the PSO framework being used a fitness function had to be developed. This fitness function needed to be a function that could determine if one solution was better than another and by how much. It could specify how good a solution is by filling in a floating point value inside the particle structure when the fitness calculation function was called. A better solution is interpreted to be one that has a lower fitness value than another.

D. Pick Space

It was mentioned previously that a 72-dimensional space was being used to solve the Sudokus. This Sudoku solver is using a row based solution system, where each row is viewed as a permutation. Each row can contain the digits one through nine in any order, but each digit can only occur once.

This can also be interpreted as a sequence of selections from an ordered set of the numbers one through nine. It would start with an ordered set and a empty row as shown in figure IV-D.

Available: [1 2 3 4 5 6 7 8 9]
Row: [ ]

Fig. 1. No Pick Complete

Next there are a set of "picks" that select the order in which the numbers will appear in the row. If the first "pick" is assumed to be 5, the 5 would be moved from the available set to the row, this is shown in figure IV-D.

Available: [1 2 3 4 6 7 8 9]
Row: [5]

Fig. 2. One Pick Complete

If the next pick is also 5 then the 6 is now moved from the available set to the row set, finally shown in figure IV-D.

Available: [1 2 3 4 7 8 9]
Row: [5 6]

Fig. 3. Two Picks Complete

This process continues until the row set is filled and the available set is empty. This will only take 8 picks as once the last pick is reached, there is only one number available.

It can be observed that in this "pick" space the size of each dimension decreases with each pick, starting with 9 and decreasing down to the known pick with a size of 1. In this implementation if a pick was outside the space then the nearest number would be chosen, one if it is too low and the max of the dimension if it is too high. When these picks are extended to be 8 dimensions for each of the 9 rows the problem has a 72-dimensional search space.

This "pick" space turned out to be a good intuitive space for searching Sudokus because a movement in any dimension by one unit will create a swap of two numbers. The goal of the fitness function is to make it so puzzles similar to the solution will be only a series of swaps away from the actual solution.

E. Fitness Function Basis

The first implementation was the most intuitive implementation of the fitness function that could be found. The first system was very generic in that location in the puzzle did not affect fitness at all. It would first generate the candidate solution based on the location of the particle, then it would overwrite any constraint in the puzzle. Once this was done it would have an array representing an attempted solution to the puzzle as shown below in figure IV-E. The red numbers show constraints that have taken the place of whatever value was chosen based on particle location.

To calculate the fitness each of the duplicates were counted Since any number should occur only once in each row, column or box, any duplicate can be perceived as a problem in the puzzle and therefore the more duplicates the less fit a solution is. So every column, row, and box is scanned for duplicates and the fitness is set to the sum

9				6	3	4	7	
				4				1
1				8		2		6
		3					8	9
	1		9	5	6		4	
4	9					1		
3		5		2		9		8
6				7				
	4	1	8	9				3

Fig. 4. Attempted Solution

of every duplicate. The missing numbers could have been analyzed as well; however these are actually equal to the number of duplicates, as for every duplicate that exists there is one number that was overwritten and is missing.

This fitness function performed relatively well overall, it minimized to puzzles that had very few duplicates. The problem that arose was that sometimes a puzzle with few duplicates could be far from the solution. The solution space seemed to be littered with local minima for most problems.

F. Modified Fitness Function

To increase the effectiveness of the fitness function, several aspects of the sudoku construction were observed. The first thing that lead to modifying the fitness function was that there were a large number of candidate solutions that had the same fitness value. There needed to be some way to determine if one of the solutions was better than another when they had drastically different structures.

The first modification was to make boxes to the left count more than boxes to the right. The reasoning for weighting more importance on left sided boxes is that the "pick" space chooses from the left to the right. Weighting the left side of the puzzle more heavily encourages the algorithm to fix the first dimensions of the picks in each row before moving onto the smaller dimensions on the right.

The second modification was to add a component to the fitness for having an incorrect constraint. Although the constraint boxes are overwritten to find the solution, having a wrong constraint can make it more difficult to get the correct permutation of the row. This fitness addition carried the same left-weightedness as stated in the first modification.

The final modification was to attempt to keep particles within the search space. Although being outside the search space does not cause any problems with the algorithm, it is known the actual solution will reside at a location within the range of the pick space. Because of this the fitness was increased for each dimension that was outside the bounds for that given pick.

G. Pick Space Constraints

Although the constraints push the problem in the correct direction through the fitness, there is some knowledge about the pick space locations of the true solution that can be known immediately. Figure IV-G shows only the constraints of a given puzzle that was solved, where the highlighted values are ones that have known pick constraints.

5	3	8	6	9	4	4	3	2
3	9	2	7	3	7	6	4	5
8	2	7	9	2	3	9	8	6
6	8	9	5	3	2	1	1	2
9	3	2	6	8	4	7	9	4
6	7	3	9	6	1	6	8	3
9	8	3	1	7	6	5	4	7
3	7	5	1	9	7	4	2	9
2	5	1	4	4	3	8	9	7

Fig. 5. Known Pick Space Values

This figure shows a number of boxes as highlighted, but most of them can be covered by a few simple rules. When a constraint of the puzzle is in the first column, the pick space equivalent can be determined. When a constraint of the puzzle is a 1 or a 9, then the pick is either 1 or the max of the dimension accordingly. When both of the last two columns are constraints of the puzzle, then the pick equivalent is 2 if the first is bigger than the right and 1 if not.

V. Results

At first the PSO based Sudoku solver did not work at all. Most PSO parameters showed it unable to even come up with good attempts at solving the puzzle. However through trial and error a set of parameters giving good performance were found. Since the space was so complicated a low swarm best weight was used (0.05), a high local best weight was used (1.0), with a fairly normal momentum of .9.

These were the first parameters that were able to solve the Sudoku successfully. The region around these parameters was analyzed and as described later in this section these were very close to the most optimum parameters for finding solutions

Once a solution had been found, most of the tests were run with a max number of iterations set to 1.65 Million. These tests took a half hour each on the CUDA-based PSO framework used. These same tests when run using the computers CPU, rather than GPU took slightly over 24 hours, making the speedup approximately 48. It is noteworthy that the runs described in this paper utilized the GPU processors of a GTX 260 for approximately a week of total run time. These same runs would have taken approximately a year to run on a conventional computer.

A. Solved puzzles

The first puzzle solved was the puzzle shown in IV-G. It also solved the following puzzle in figure V-A.

5	3	8	6		4			
	9	2		3	7	6	4	5
			9	2			8	
						1		2
9								4
6		3						
	8			7	6			
3	7	5	1	9		4	2	
			4		3	8	9	7

Fig. 6. Another Sudoku Puzzle

B. Effect of parameters

It was difficult at first to find any PSO parameters that would solve the Sudoku puzzles given. Because of this the effect of the PSO parameters was studied. A series of tests, each with a maximum of 1.65 Million iterations, were run on various changes in the PSO parameters. Four random seed values were chosen, for each seed a specific set of starting particles exists. Each of the four seeds were used on every set of parameters tested.

The results showed that having the momentum at 0.9 was very much a perfect spot for solving sudoku. Changing the momentum in either direction seriously interfered with the ability to find a solution. Figure V-B shows the average ending fitness compared to the momentum.

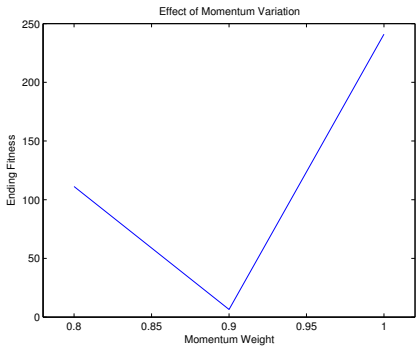


Fig. 7. Effects of Momentum Coefficient

The local best weight and the swarm best weight were varied from 0.85 to 1.15 and from 0.05 to 0.15 respectively. Through a series of previous runs it was determined that these were the areas that the algorithm could be successful. These parameters proved to be much less sensitive than the momentum. The low fitness range of the results suggest that given enough time it is possible that any of these sets of parameters could solve the puzzle. Figure V-B below shows the final fitness vs the local best weight and the swarm best weight.

Fig. 8. Effects of PSO Weights

There is an upward trend as both of the weights increase. This could suggest that the algorithm is simply taking more time and it will eventually find a solution, or it could suggest that if the weights increase too much it will be unsuccessful in finding a solution. In other isolated tests larger weights did not perform well and often would not converge or would converge in very high local minima not near the solution.

C. Random Addition

Even a small amount of testing showed that this was very clearly a very bumpy search space, full of local minima. This was shown in the last section by the heavy weighting upon particles local best and light weighting on swarm best locations. Despite this weighting some tests revealed it was possible for the algorithm to come very close to finding a solution without actually finding it.

The solution to this problem was a random addition. This had no biological basis, and the random numbers had such a small coefficient that it had virtually no affect on the running of the algorithm. However despite its small nature it did help in scenarios where the entire swarms were stuck at a single local minima. The random addition to all particles location would give them some momentum, allowing more searching of the space.

VI. Conclusions

This implementation successfully solved multiple Sudoku puzzles and has yet to be attempted on a puzzle that it did not eventually solve. The speedup of the CUDA-based PSO framework that was used was instrumental in the success of this project, as even to run 1.6 Million iterations on the GPU still took on the order of 30 Minutes to complete.

Due to the bumpy nature of the parameter space a much smaller swarm best location weight than usual must be used to solve the Sudoku puzzles using PSO. This problem also proved to be extremely sensitive in variation in the momentum coefficient.

VII. Acknowledgements

This work was partially funded by NSF grants EAR 1027809, OIA 0619430, DRL 0737583, CCF 0754951, EPS 0918018, EPS 0904155, and NIH grant R01 HL092926.

References

[1] J. Monk, K. Hanselman, R. King, Y. Zhu, and B. Segee, "Parallel multi-swarm particle swarm optimization using cuda." Submitted for Publication, 2011.

[2] T. Mantere and J. Koljonen, "Solving, rating and generating sudoku puzzles with ga." IEEE, 2007.

[3] C. Boyer, "Sudoku's fench ancestors," 2007.

[4] A. Moraglio, C. Chio, J. Togelius, and R. Poli, "Geometric particle swarm optimization," in *Journal of Artificial Evolution and Applications*, vol. 2008. IEEE.

[5] J. Jilg and J. Carter, "Sudoku evolution." IEEE, 2009.

[6] R. Eberhart and J. Kennedy, "Particle swarm optimization," in *IEEE international Conference on Neural Networks*, vol. 4. IEEE Press, 1995, pp. 1942– 1948.

[7] T. Xiang, J. Wang, and X. Liao, "An improved particle swarm optimizer with momentum." IEEE, 2007.

[8] Y. Zhou and Y. Tan, "Particle swarm optimization with triggered mutation and its implementation based on gpu," GECCO. ACM, Jul. 2010.

[9] L. Mussi, Y. Nashed, and S. Cagnoni, "Gpu-based asynchronous particle swarm optimization," GECCO. ACM, Jul. 2011.

[10] L. Venneschi, D. Codecasa, and G. Mauri, "An empirical comparison of parallel and distributed particle swarm optimization methods," GECCO. ACM, Jul. 2010.

[11] S. Solomon, P. Thulasiraman, and R. Thulasiram, "Collaborative multi-swarm pso for task matching using graphics processing units," GECCO. ACM, Jul. 2011.

SESSION

WORKSHOP ON MATHEMATICAL MODELING AND PROBLEM SOLVING, MPS

Chair(s)

Prof. Minoru Ito

GAROP: Genetic Algorithm framework for Running On Parallel environments

Tomoyuki Hiroyasu*, Ryosuke Yamanaka†, Masato Yoshimi‡ and Mitsunori Miki‡
*Faculty of Life and Medical Sciences, Doshisha University, Kyoto, Japan
†Graduate School of Engineering, Doshisha University, Kyoto, Japan
‡Faculty of Science and Engineering, Doshisha University, Kyoto, Japan

***Abstract*—In this research, a Genetic Algorithms framework for Running On Parallel environments, which is named GAROP, is proposed. The GAROP provides the library for a parallel processing, so that users should only describe codes for genetic algorithms (GA) programs, utilizing the library implemented for the part requiring a parallel processing. In the GAROP framework, GA research provides only program codes which are concerned with GA algorithm and GAROP library supports other codes which are concerned with parallel processing. The advantage of using GAROP is to increase the user's productivity by making it possible to develop the program, which can execute a parallel processing. In this paper, the broad description of the GAROP is provided, and the development of the GAROP, corresponding multi-core CPU and GPU environments, is described. The libraries are implemented with GA which finds quasi-optimum solutions using meta heuristics, and its productivity and its parallelism are evaluated. As a result, only adding four descriptions to the program, the acceleration of the processing speed is confirmed in both of the environments; 5.26 times speed-up on multi-core CPU, and 3.0 times speed-up on GPU.**

I. Introduction

Several types of genetic algorithms (GA) are applied to solve optimization problems and some of them are large-scale optimization problems. One of the problems confronted, when using the GA, is that the excessive amount of computing time is required. It may be difficult for some problems to be solved within a realistic time. To solve this problem, the amount of computation itself should be reduced, or the processing should be accelerated. GA attains to a global search, using multipoint searches by many candidate solutions. It requires much iteration to find a solution, and results in high calculation cost. As GA searches solutions maintaining many candidate solutions, it implicitly has parallelism. As the architectures of calculators, on the other hand, hardware having various architectures has been prevailed, such as PC clusters, multi-core processor, and GPU. Therefore, to obtain environments of parallel calculations is not as hard as ever. Although an environment is easily obtainable, the programming skills are required to bring out its performance efficiently, such as the programming for heterogeneous processors, the architectural optimization for hierarchical memories, and the programming which overlaps connection and calculation to achieve the scalability. In addition, these parallel architectures have the different configurations. Thus, even using the same algorithms, it is necessary to prepare different implementation codes suitable for different parallel architectures. This complicated and disturbing programming lacks productivity. In this research, the Genetic Algorithms framework for Running On Parallel environments, which is named GAROP, is proposed. The purpose of the GAROP is to increase the user's productivity by reducing the processing time without the specific knowledge regarding parallel architecture and parallel processing. Implementing the master-slave model, the GAROP enables to execute any logical models. In this paper, the libraries for the GAROP are implemented using C language. Target parallel processing architectures are multi-core CPU and GPU. As one of the GA technique, Simple GA is implemented using the libraries, and it is evaluated in terms of an amount of codes and execution time.

II. Background

A. Genetic Algorithms

GA is a multipoint search algorithm with many candidate solutions and generate-and-test algorithm. GA is powerful algorithm in all kinds of research field, because GA makes it possible to search globally, and it does not require continuity and differentiability of target problems [1]–[5]. Under GA, a combination of variables, which is to be a candidate solution, is termed an individual. GA searches suboptimal solutions with a group of individuals. Figure 1 shows a general flow of GA. The first group of individuals is generated randomly. Generated individuals are evaluated on their objective function. Then, iteration searches are started. Parent individuals are selected from the group of individuals. Individuals are generated from selected parents as candidate children by genetic calculations, such as crossover and mutation. Generated individuals as candidate children are evaluated on their objective function. Selecting from these individuals, a group of individuals for next generation is created. Presently, various types of GA algorithms have been proposed, and these algorithms are applied to real-world problems. However, an excessive amount of computation is required to find suboptimal solutions in large-scale problems. Thus, it may be difficult for some problems to be solved within a realistic time, so that the amount of computation should be reduced, or the processing should be accelerated.

B. Parallel Genetic Algorithms

Since GA processes a search by iterative execution of an excessive amount of samplings on multiple candidate solu-

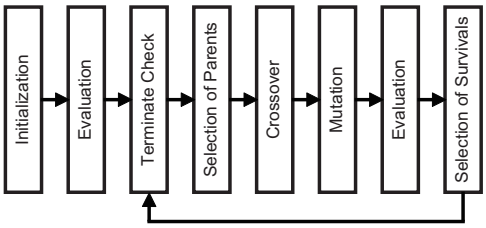


Fig. 1. A flowchart of GA.

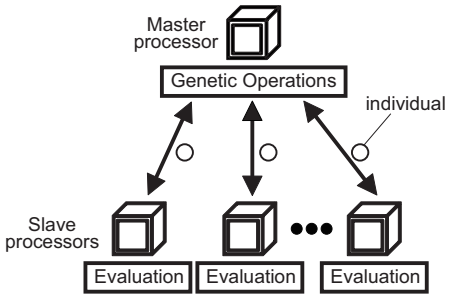


Fig. 2. A master-slave model.

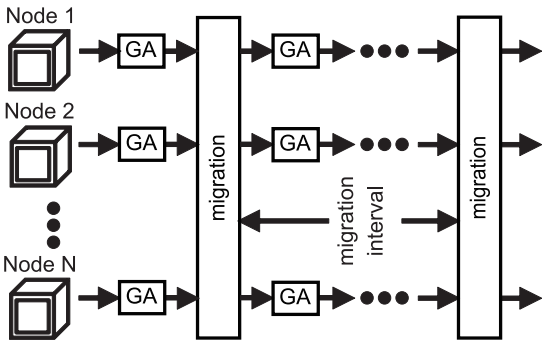


Fig. 3. An island model.

tions, it can be accommodated to parallelism. Thus, many methods of parallelization have been proposed for GA [6]–[12]. In this section, basic parallelizing methods of GA are described. There are two principal types of parallel GA. One is parallel processing in a population. This method has the same characteristic as serial GA. The other is to split a population into multiple subpopulations. Today, the latter method is frequently used, because it has the higher parallelism than the former method. There are GA methods performing very efficient parallelism, which combine the both methods. On changing the method of parallel GA, it is important to consider that it may happen to change the amount of calculations and the accuracy of solutions. That brings to the point that Parallel GA has the following two meanings.

- Parallel algorithm for increasing search performance
- Parallel implementation for reducing execution time

For example, Pospichal [13] has proposed the distributed population GA based on GPU, and achieved a high efficiency. Although this method has achieved high efficiency, GA other than the distributed population GA cannot be implemented, since GA and parallel implementation are inseparable Also, it is difficult to implement this method into architecture other than GPU. The most basic parallel models are introduced as the following.

1) The Master-Slave Model: Under GA, there is a tendency that the time consumed on evaluation calculations assumes a large share of total execution time, and the tendency strengthens as the complexity of the problem is increased. Then, the master-slave model accommodates this tendency based on the general idea of parallelization. Under the master-slave model, all the operations except evaluations are executed by a master processor. A master processor sends individuals, which are to be evaluated, to slave processors. Slave processors execute evaluating calculations on these individuals, and return the results to the master processor. Figure 2 shows the flow of the master-slave model. It is considered that this model is inferior to coarse-grained model, because this model requires relatively much communication, and a CPU is requisite as a master processor. The purpose of this model is to reduce the execution time, so it cannot increase the searching performance compared to a serial algorithm.

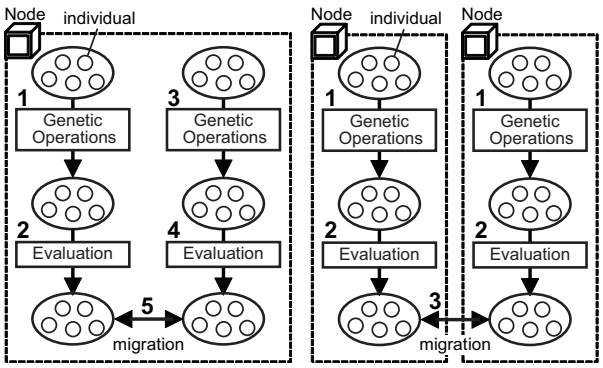
2) The Island Model: This model splits up a population into multiple sub populations and executes searching within each sub population. Then, it transports some individuals in a sub population to the other sub population. This operation is called the migration. Figure 3 shows the flow of the island model. Since this model makes communications between nodes only at the migrations, this model utilizes computational resources effectively. This model reduces its execution time and changes the performance of the search compare to a serial algorithm.

C. Problems

As previously mentioned, parallel GA has two objectives; those are to improve searching performance and to reduce execution time. Some of parallel GA models are depend on specialize particular parallel environment and these cannot be performed on other parallel environments. These types of GA models can use the calculation resources fully and high effectiveness. However, most of parallel calculation resources have difference architecture. Thus, when GA is tried to apply to other parallel architecture, GA researchers have to implement their algorithms for new architecture. To know configurations of various architectures and to implement suitable GA are the heavy burden on GA researchers. With these defects, even research have good parallel environments, it may take time to implement their algorithm.

III. Systematization of Parallel Model

As previously mentioned in chapter II, the expression of "parallel GA" has two models; one is a parallel algorithm to increase searching performance, and the other is a parallel implementation to reduce the execution time. These two models have to be distinguished clearly. In this research, these two models are defined as the followings:



(a) A serial implementation. (b) A parallel implementation.

Fig. 4. Two island models as the logical model

- The logical model: parallel algorithm to increase searching performance
- The implementation model: a parallel implementation to reduce the execution time

Figure 4(a) shows GA adopted an island model as a logical model and a serial model as an implementation model. Figure 4(b) shows GA adopted and island model as a logical model and also as an implementation model. In Figure 4, the number means the orders of processing. The logical model is a model to execute parallel searching, but it is capable to execute serial processing. Additionally, the logical model may be confined by a limitation imposed by the implementation model. For example, if the island model is adopted as the implementation model, the Simple GA cannot be adopted as the logical model. Once the implementation model is provided, only users have to do is to consider and implement the logical model. Thus, users are able to develop any algorithms without the limitation of architectures.

IV. GAROP

The GA framework for Running On Parallel environments (GAROP) is a framework in which any GA can be executed in various parallel environments. The purpose of the GAROP is that users can execute parallel processing, with the master-slave model as the implementation model, without having special techniques for parallel programming. The users of the GAROP are presumed as the developers of GA. Once the implementation for parallel processing is provided, users could receive the benefit of parallel processing, keeping the same level of productivities as sequential programs. Under the GAROP, users construct any logical models and implement some parts other than the evaluation part. Users designate the template suited to each parallel environment, and combine the template and the codes for evaluation of the problem, so that the evaluation part is implemented. The use of this template leads to hide the special communications and the implementation of scheduling for evaluation tasks. Thus, users can execute GA under parallel environments without having knowledge of communication and schedulers appropriate for parallel environments.

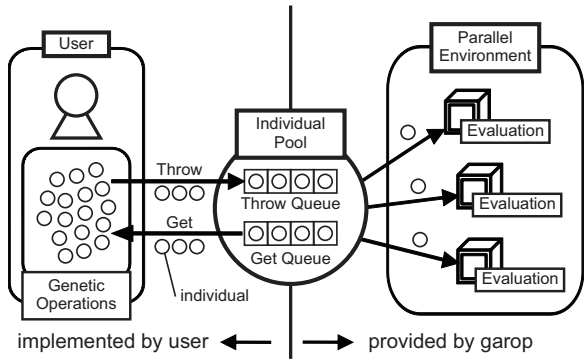
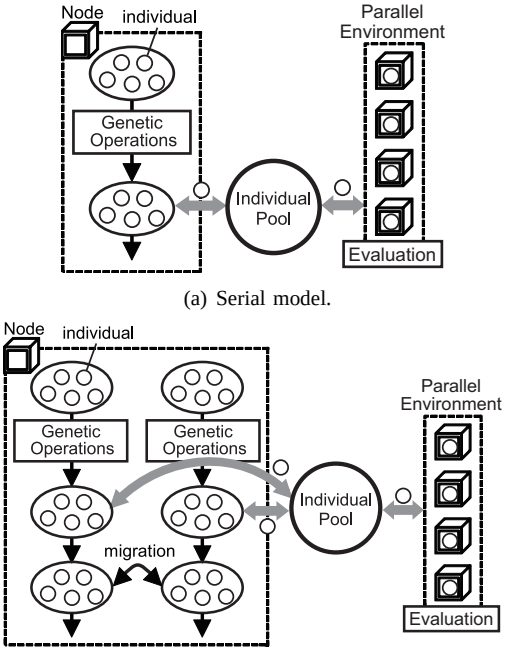


Fig. 5. Overview of the GAROP.



(a) Serial model.

(b) Island model.

Fig. 6. GA Examples with GAROP.

A. Requirements

To achieve the purpose as precious described, the GAROP is expected to meet the following requirements:

- Productivity of users
  In order to achieve the same level of productivity as the sequential programs, the framework should be provided as the form of libraries which call up a function group.
- Versatility of parallel environments
  Users should be able to use various parallel environments by using common descriptions.
- Independent implementation model
  In order to correspond to any GA, the implementation model which is implemented independently of algorithms is required.

TABLE I
FUNCTIONS PROVIDED BY GAROP.

Name	Action
Initialize	prepare parallel environments and the Individual Pool.
Throw	throw an individual to the Individual Pool.
Get	get an individual from the Individual Pool.
Finalize	free memories used by the Individual Pool and disconnect parallel environments.

B. Design of the GAROP

Figure 5 shows the overview of the GAROP. GAROP introduces the concept of the Individual Pool as an interface to link users to parallel environments. The Individual Pool is an archive and stores individuals which should be evaluated in parallel automatically. Under this design, any GA models can be executed in parallel without change searching performance. In Figure 6, the concepts of serial model and island model are demonstrated. In the same way, other parallel models can be performed using the GAROP.

1) The Individual Pool: The Individual Pool consists of two queues as shown in Figure 5. One of these queues is the "throw queue", which stores thrown individuals, and the other queue is the "get queue", which stores evaluated individuals. As soon as individuals are stored in the throw queue, they are sent to calculation resources and evaluated. Evaluated individuals are stored in the get queue. Users throw individual which should be evaluated to the Individual Pool one by one. Users can get the evaluated individual from the get queue, whenever they need. The Individual Pool is a useful concept which makes it possible to execute evaluating calculations and other processing simultaneously.

2) Programming Interface: The GAROP provides 4 functions as shown in Table I. This won't be changed even though the environment of the parallel computation is changed. However, the types of four functions and arguments vary depending on the environment.

3) The Template of evaluation: It is impossible to provide implementations of every single evaluation, because the evaluation depends on the problem to be solved. In the GAROP, users implement the evaluation part. This means that the implementation of the evaluating calculation exists on the memory of the master machine. However, the evaluation is executed by slave processors. Therefore, the evaluation part must be handed over from the master machine to the slave processor. It is realistic to hand over using a form of a function, which is easy to describe. In order to figure out how much memory region is needed, the types and the number of arguments and the type of return value should be known, when the function is handed over. The GAROP provides the template of an evaluate function as the following. Users can solve any problems by limiting the arguments and the return value of the function.

```
void evaluate(unsigned char* indata,
              unsigned char* retdata);
```
- indata: individual data
- retdata: evaluated individual data

This way of description is suitable for the C language. The important thing is to allocate an individual to the argument and to allocate another individual to the return value. In order for a description of an individual to be free, it employs unsigned char as a pointer. The template of the evaluating function varies depending on each of the parallel environment and the language used.

C. How to run algorithms with the GAROP

The libraries to substantiate the GAROP are provided in the form of source codes. Multiple libraries are prepared for each of the execution environments, such as compilers and languages. The user's flow is shown as the following.

- Obtaining the library (source codes) corresponding to the executing environment
- Implementing evaluate function using the templates
- Implementing GA with API of the library
- Compiling source codes
- Placing executable file into calculation resources
- Executing

List 4 is an example of evaluate function using a template. List 1 is an implementation of the serial model with the GAROP. List 2 is an implementation of the island model with the GAROP. Like this way, not only the master-slave model but also the island model can be implemented. Using the GAROP, other models such as cellular model can be implemented in the same way. Under the GAROP, the parallel environment is set up by users. For example, if users use PC cluster with MPI, users prepare cluster by themselves and give a description of machine file etc.

V. IMPLEMENTATIONS OF LIBRARIES FOR THE GAROP

The libraries are implemented to get individuals from Individual Pool and to evaluate in parallel. The environments implemented in this paper are the multi-core CPU by pthread of the C language and the GPU by CUDA. The multi-core CPU is a shared memory environment, and the GPU is a distributed memory environment. Policies of these environments are as the following.

A. Multi-core CPU

The characteristics of multi-core CPU are having not many cores and having a shared memory environment. Considering these two characteristics, the library is implemented with a constitution shown as Figure 7. A thread is regarded as a calculation resource and assigned as a slave processor. Each thread monitors the throw queues. When the throw queue has one or more data, each thread gets an individual and executes evaluating calculations. Arguments of an initialize function are number of threads, size of an individual, and pointer of an evaluate function.

List 1. Serial model with the GAROP.

```
1  population = InitPopulation();
2  Initialize(); // initialization of framework
3  FOR j = 0 to generation limit DO
4    FOR i = 0 to population num DO
5      // throw individuals to GA Pool
6      Throw( population[i] );
7    ENDFOR
8    FOR
9      // get individuals from GA Pool
10     Get( population[i] );
11   ENDFOR
12   selection( population );
13   crossover( population );
14   mutation( population );
15 ENDFOR
16 Finalize(); // finalization of framework
```

List 2. Island model with the GAROP.

```
1  population1 = InitPopulation();
2  population2 = InitPopulation();
3  Initialize(); // initialization of framework
4  FOR j = 0 to generation limit DO
5    FOR i = 0 to population num DO
6      // throw individuals to GA Pool
7      Throw( population1[i] );
8      Throw( population2[i] );
9    ENDFOR
10   FOR
11     // get individuals from GA Pool
12     Get( population1[i] );
13     Get( population2[i] );
14   ENDFOR
15   selection( population1 );
16   selection( population2 );
17   crossover( population1 );
18   crossover( population2 );
19   mutation( population1 );
20   mutation( population2 );
21   IF j % 10 == 0 THEN
22     migration()
23   ENDIF
24 ENDFOR
25 Finalize(); // finalization of framework
```

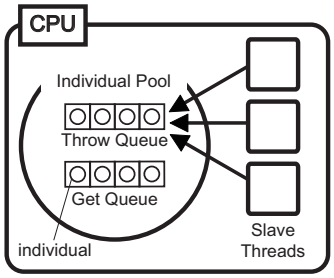


Fig. 7. An implementation of the GAROP for multi-core CPU.

B. GPU

The GPU has many cores and has a distributed memory environment. As the GPU has a distributed memory environment, the library is implemented with a constitution shown as Figure 8. In CPU, there are main thread which run GA and sub thread which communicate data to GPU. Arguments of initialize function are number of blocks and thread, size of an individual, and number of individuals that are processed at a kernel function call. Individuals sent to GPU are stored in the constant memory. Each CUDA thread acquires individual information from constant memory, and writes evaluated individual data to global memory.

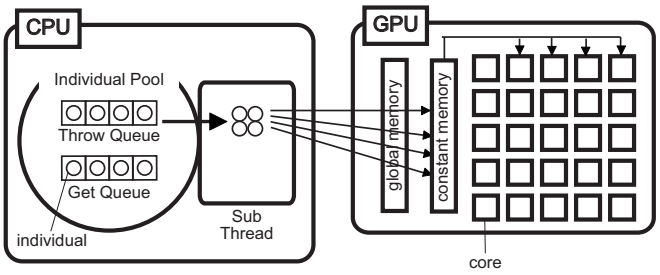


Fig. 8. An implementation of the GAROP for GPU.

TABLE II
PARAMETERS OF THE SIMPLE GA

population size	64
choromosome length	64
size of an individual	68 bytes
max generations	100
optimization problem	onemax

VI. EVALUATION

The following is the evaluation of a Simple GA [14] implemented using the GAROP, in terms of productivity and parallelism. Table II shows a parameter of the Simple GA. List 3 shows a description of an individual in this evaluation. In this experiment, onemax problem is iterated 100,000 times to mimic a large-scale problem. List 4 shows a function on the evaluation. In this GA, the evaluating calculation takes up more than 99 % of total execution time. The maximum number of parallelism is 64, since the population size is 64. Thus, number of individuals to be calculated at a kernel function call is 64 in the GPU library.

A. Environments

The architectures to be evaluated in this experiment are multi-core CPU and GPU. Table III shows the specifications of

List 3. an implementation of an individual

```
1 typedef struct __individual {
2   char* chromosome;
3   int fitness;
4 } Individual;
```

List 4. an evaluate function used in this experiment

```
1  void evaluate( unsigned char* indata,
2                 unsigned char* retdata ) {
3    int i, j;
4    int sum = 0;
5    Individual* individual = (Individual*)indata;
6    for( j = 0; j < 100000; j++ )
7      for( i = 0; i < CHROMOSOME_LENGTH; i++ )
8        sum += individual->chromosome[i];
9    individual->fitness = sum;
10   retdata = indata;
11 }
```

TABLE III
SPECIFICATIONS OF A MASTER MACHINE

OS	Debian 4.1.2
memory	6 GB
CPU	Intel Xeon W3530 2.80 Ghz
# of physical cores	4
# of logical cores	8

TABLE IV
SPECIFICATIONS OF TESLA C2050

total amount of global memory	2.68 GB
number of multiprocessors	14
number of cores	448
total amount of constant memory	65536 bytes
total amount of shared memory per block	49152 bytes
warp size	32
clock rate	1.15 GHz

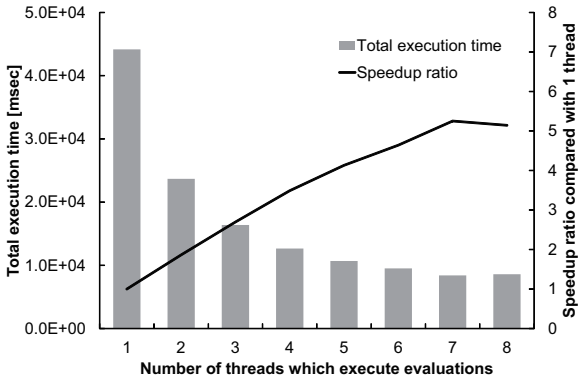
the machine used in this evaluation. This machine is mounted Tesla C2050 as shown in Table IV.

B. Results

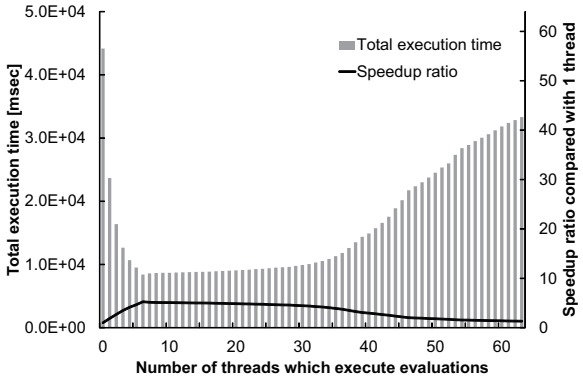
Figure 9 shows results of multi-core CPU, and Figure 10 shows results of GPU, respectively. List 5 shows the source codes which described by users on multi-core CPU. List 6 shows the source codes which described by users on GPU. List 5 and List 6 show that these descriptions are common among different architectures. It is verified that descriptions for parallel processing can be completely hidden, even though these descriptions are essentially required on parallel processing. Figure 9(a) shows that the execution time reduces when the number of threads is 1 to 7. Figure 9(b) shows that the number of threads that had the least processing time is 7, and it is improved 5.26 times compared to the situation when a thread is processed. When the number of thread is 8 or more, execution time takes longer. Figure 10 shows that the number of threads that had the least processing time is 64, and it is improved 50.01 times compared to the situation when a thread is processed. When the numbers of thread are 2, 3, 4, 6, 8, 16, 32 and 64, the execution time is extremely reduced.

VII. DISCUSSION

The C language and CUDA are the similar languages, but methods of implementation in parallel are substantially different. C is used for multi-core architecture and CUDA is used for GPU architecture. Usually, users need the parallel programming knowledge of C and CUDA for these architectures. However, using the GAROP framework and its library, users need not to prepare the codes for parallel processing. Both of the codes which GA users prepare can be used as the common descriptions. Therefore, the productivity of coding is increased using the GAROP. Reviewing the result of multi-core CPUs, when the number of threads is 8, the execution time is the shortest. Among the 8 threads, 7 of them are the threads for the slave processors and the rest is the main thread for GA operations. The calculation server for this experiment



(a) The number of threads (1 to 8).



(b) The number of threads (1 to 64).

Fig. 9. Speedup and execution time on multi-core CPU depending on number of threads.

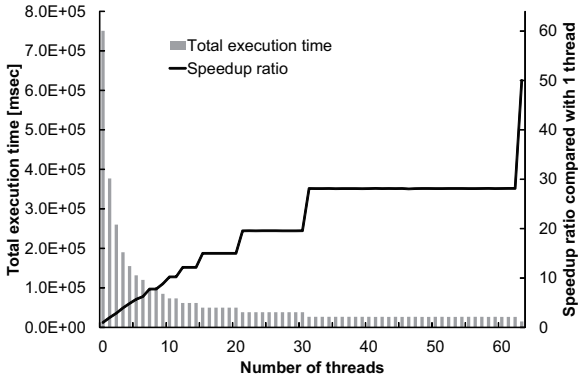


Fig. 10. Speedup and execution time on GPU depending on number of threads.

has 8 logical cores. Therefore, when 7 threads are used for the slave processors, all of the logical cores are occupied. When 8 or more threads are used, the number of threads is more than the number of logical cores, so that it prevents the speed up. When the number of threads is more than the number of cores, it may happen that CPU has to switch executing threads one after another, and this operation of switching threads produces the waiting time. In the experiment in GPU, the execution time is the shortest with 64 threads. Tesla C2050, which is the GPU

List 5. Simple GA with a library for pthread in C.

```
Individual population[POPULATION_SIZE];
// create population
InitPopulation( population );
// initialization of framework
Initialize( sizeof(Individual), THREAD_NUM, evaluate );
for( i = 1; i <= MAX_GENERATION; i++ ) {
    // throw individuals to GA Pool
    for( j = 0; j < POPULATION_SIZE; j++ )
        Throw( (unsigned char*)&population[j],
               sizeof(Individual) );
    // get individuals from GA Pool
    for( j = 0; j < POPULATION_SIZE; j++ )
        Get( (Individual*)&population[j],
             sizeof(Individual) );
    population = selection( population );
    crossover( population );
    mutation( population );     }
// finalization of framework
Finalize();
```

List 6. Simple GA with a library for GPU in CUDA.

```
Individual population[POPULATION_SIZE];
// create population
InitPopulation( population );
// initialization of framework
Initialize( sizeof(Individual), BLOCK_NUM,
            THREAD_NUM, POPULATION_SIZE );
for( i = 1; i <= MAX_GENERATION; i++ ) {
    // throw individuals to GA Pool
    for( j = 0; j < POPULATION_SIZE; j++ )
        Throw( (unsigned char*)&population[j],
               sizeof(Individual) );
    // get individuals from GA Pool
    for( j = 0; j < POPULATION_SIZE; j++ )
        Get( (Individual*)&population[j],
             sizeof(Individual) );
    population = selection( population );
    crossover( population );
    mutation( population );     }
// finalization of framework
Finalize();
```

used in this experiment, has 448 cores. Thus, it can use 64 cores for 64 threads. From the Figure 10, it is observed that the stagnation of speed up is existed from 17 to 31 threads and from 33 to 63 threads. The reason of this stagnation is that the population size 64 is not the multiple of these numbers. Because of this reason, the fraction of the individuals is existed and the fraction itself should be calculated, too. This takes time and leads to the stagnation.

VIII. Conclusions

In this paper, the GAROP which is a parallel environment framework for evolutionary computation is proposed. The GAROP is the framework where the logical model and the implementation model are distinguished, and the users prepare their algorithms as the logical model and the implementation model is prepared by systems. Thus, users can implement any type of logical model on parallel environment using the GAROP. In the GAROP, user's evolutionary algorithms are performed in parallel as master-slave model. Users implement their GA operations in the master and the evaluation part is implemented in the slave using the provided template. With the provided libraries and the implemented codes, the application, which is worked on several types of parallel environment, is compiled. In this paper, the concept and the flow of the GAROP are described and the libraries for two types of parallel environments are implemented; those are multi-core CPUs and GPUs. Using the GAROP and the libraries, Simple GA is implemented and the productivity of users and parallelism are evaluated. From the results, GA applications which can be worked on parallel environments are implemented using four types of functions which are provided by the GAROP. At the same time, execution time is also reduced.

In the future work, further discussion should be held for not only Simple GA but also for other evolutionary computation algorithms. In this paper, the libraries for multi-core CPUs and GPUs are implemented. Other types of libraries for other parallel environments will be prepared. At the same time, the productivity discussions of the GAROP should be performed with researches who are working on evolutionary computation fields.

References

[1] B. Chakraborty, T. Maeda, and G. Chakraborty, "Multiobjective route selection for car navigation system using genetic algorithm," in *Soft Computing in Industrial Applications, 2005. SMCia/05. Proceedings of the 2005 IEEE Mid-Summer Workshop on*, june 2005, pp. 190–195.

[2] R. Ruiz, C. Maroto, and J. Alcaraz, "Two new robust genetic algorithms for the flowshop scheduling problem," *OMEGA, The International Journal of Management Science*, vol. 34, no. 5, pp. 461–476, 2006.

[3] P. C. Chang, H. J. Chang, and W. C. Yuan, "Adaptive multi-objective genetic algorithms for scheduling of drilling operation in printed circuit board industry," *"Applied Soft Computing*, vol. 7, no. 3, pp. 800–806, 2007.

[4] C. Poloni, A. Giurgevich, L. Onesti, and V. Pediroda, "Hybridization of a multi-objective genetic algorithm, a neural network and a classical optimizer for a complex design problem in fluid dynamics," *Computer Methods in Applied Mechanics and Engineering*, vol. 186, no. 2-4, pp. 403–420, 2000.

[5] B. Ombuki, B. J. Ross, and F. Hanshar, "Multi-Objective Genetic Algorithms for Vehicle Routing Problem with Time Windows," *Applied Intelligence*, vol. 24, pp. 17–30, 2006.

[6] T. Starkweather, D. Whitley, and K. Mathias, "Optimization using distributed genetic algorithms," in *Parallel Problem Solving from Nature*, ser. Lecture Notes in Computer Science, Schwefel, Hans-Paul and Männer, Reinhard, Ed. Springer Berlin / Heidelberg, 1991, vol. 496, pp. 176–185.

[7] H. Mühlenbein, "Parallel genetic algorithms, population genetics and combinatorial optimization," in *Parallelism, Learning, Evolution*, ser. Lecture Notes in Computer Science, J. Becker, I. Eisele, and F. Mündemann, Eds. Springer Berlin / Heidelberg, 1991, vol. 565, pp. 398–406.

[8] T. C. Belding, "The distributed genetic algorithm revisited," *Proc.6th International Conf. Genetic Algorithms*, pp. 114–121, 1995.

[9] M. Miki, T. Hiroyasu, M. Kaneko, and K. Hatanaka, "A Parallel Genetic Algorithm with Distributed Environment Scheme," *IEEE International Conference on Systems, Man, and Cybernetics*, vol. 1, pp. 695–700, 1999.

[10] D. Lim, Y. S. Ong, Y. Jin, Sendhoff.B, and L. B. Sung, "Efficient Hierarchical Parallel Genetic Algorithms using Grid computing," *Future Generation Computer Systems*, vol. 23, no. 4, pp. 658–670, 2007.

[11] J. M. Li, X. J. Wang, R. S. He, and Z. X. Chi, "An efficient fine-grained parallel genetic algorithm based on gpu-accelerated," in *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, Sep. 2007, pp. 855–862.

[12] Thompson, A. Matthew and Dunlap, I. Brett, "Optimization of analytic density functionals by parallel genetic algorithm," *Chemical Physics Letters*, vol. 463, no. 1–3, pp. 278–282, 2008.

[13] P. Pospichal and J. Jaros, "GPU-based Acceleration of the Genetic Algorithm," *GPU competition of GECCO competition*, 2009.

[14] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

Rapid Feature Selection Based on Random Forests for High-Dimensional Data

Hideko KAWAKUBO, Hiroaki YOSHIDA
Graduate School of Humanities and Sciences, Ochanomizu University, Tokyo, Japan

Abstract— *One of the important issues of machine learning is obtaining essential information from high-dimensional data for discrimination. Dimensionality reduction is a means to reduce the burden of dimensionality due to large-scale data. Feature selection determines significant variables and contributes to dimensionality reduction. In recent years, the random forests method has been the focus of research because it can perform appropriate variable selection even with high-dimensional data holding high correlations between dimensionality. There exist many feature selection methods based on random forests. These methods can appropriately extract the minimum subset of important variables. However, these methods need more computation time than the original random forests method. An advantage of the random forests method is its speed. Therefore, this paper aims to propose a rapid feature selection method for high-dimensional data. Rather than searching the minimum subset of important variables, our method aims to select meaningful variables quickly under the assumption that the number of variables to be selected is determined beforehand. Two main points are introduced to enable faster calculations. One is reduction in the calculation time of weak learners. The other is adopting two types of feature selection: "filter" and "wrapper." In addition, although most present methods use only "mean decrease accuracy," we calculate the magnitude of features by combining "mean decrease accuracy" and "Gini importance." As a result, our method can reduce computation time in cases where generated trees have many nodes. More specifically, our method can reduce the number of important variables to* 0.8% *on an average without losing the information for classification. In conclusion, our proposed method based on random forests is found to be effective for achieving rapid feature selection.*

Keywords: feature selection; variable selection; random forests; Gini importance;

1. Introduction

In recent years, feature selection for dimensionality reduction is becoming increasingly important in machine learning. Feature or variable selection enables improving accuracy by excluding redundant variables and facilitates the interpretation of complex data structures as well as reduces calculation time for predictors. Therefore, variable selection for high-dimensional data plays an important role in many areas including text processing of internet documents, gene expression array analysis and combinatorial chemistry. In this paper, we propose a rapid feature selection method for high-dimensional data.

There are three types of variable selection: "filter," "wrapper," and "embedded" [1], [2]. "Filter" selects subsets of variables in a preprocessing step, independent of the chosen predictor. "Wrapper" utilizes the learning machine of interest as a black box to score subsets of variables according to their predictive power. "Embedded" performs variable selection during the training process and is usually specific to given learning machines. The random forests (RF) method [3] based on the wrapper method has been widely recognized as a practical method of variable selection. In recent years, the RF method has also been applied to feature selection for hyperspectral imagery and gene selection of microarray data [4], [5]. Furthermore, the demand for variable selection has been increasing.

The RF method has two types of variable importance measures. One involves the evaluation of "out-of-bag (OOB) errors" introduced to estimate prediction errors. Several feature selection methods using this measure have been proposed [6], [7], [8].

The other measure is derived from the Gini index and is called "Gini importance." This measure is biased toward predictor variables with many categories [9]. However, it is particularly effective with data that have a high dimensionality and small sample size [10]. There also exists a feature selection method using "Gini importance" [11].

These feature selection methods, which are extended RF methods, can appropriately extract the minimum subset of important variables. Because the RF method itself is stochastic, the subsets obtained by these methods are only a candidate of the optimal solution; moreover, if sufficient computation time is provided, these methods are attractive.

In this paper, we assume that the number of important variables to be selected is decided beforehand and propose a fast method to select meaningful variables with a high accuracy. As a result of investigating the ranking of important variables derived from various datasets by using the original RF method, we obtain the following empirical rule: the rankings drawn from two types of variable importance measures slightly differ, whereas the members of the top ranked variables are almost the same. Based on this empirical rule, we improve the original RF method and successfully reduce

computation time, especially in cases where generated trees have many nodes. In addition, in our method, the number of important variables is reduced to 0.8% on an average without losing the information for discrimination. In conclusion, our proposed method based on RF is effective to achieve rapid variable selection. The reason why our method is successful is not solved mathematically; the results obtained by our method are very interesting.

In the following section, we review RF and "Gini importance"; we explain our proposed method in section 2.

1.1 Random forests algorithm

The RF method creates multiple trees using classification and regression trees (CART) [12]. When constructing a tree, the RF method searches for only a random subset of input variables at each splitting node and the tree grows fully without pruning. The RF method is recognized as a specific instance of bagging.

Random selection of variables at each node decreases the correlation among trees in a forest, thus forest error rate decreases. The random subspace selection method has been demonstrated to perform better than bagging when there are many redundant variables contributing to discrimination among classes [13], [14], [15].

The computational load of the RF method is comparatively light. The computation time is on the order of $ntree\sqrt{mtry}\ n\log n$, where $ntree$ is the number of trees, $mtry$ is the number of variables used in each split, and n is the number of training samples [3], [4].

In addition, when a separate test set is not available, an OOB method can be used. For each newly generated training set, one-third of the samples are randomly excluded; these are called OOB samples. The remaining (in-the-bag) samples are used for building the tree. For accuracy estimation, votes for each sample are counted every time a sample is included among OOB samples. A majority vote determines the final label. The OOB error estimates are unbiased in many tests [3]. The number of $mtry$ is defined by a user, and it is insensitive to the algorithm.

The RF algorithm (for both classification and regression) is as follows:

1) Draw $ntree$ bootstrap samples from the original data.
2) For each bootstrap sample, randomly sample $mtry$ predictors (variables) at each node, grow an unpruned classification or regression tree, and choose the best split among these variables (rather than choosing the best split among all variables).
3) Predict new data by aggregating the predictions of $ntree$ trees (i.e., majority vote is used for classification, average is used for regression).

Based on training data, an error rate estimate can be obtained as follows:

1) At each bootstrap iteration, predict test data not in the bootstrap sample (what Breiman calls "out-of-bag" or OOB data) using a tree grown with the bootstrap sample.
2) Aggregate the OOB predictions. Calculate their error rate, and call it OOB error rate estimate.

The RF method performs efficiently for large datasets and can handle thousands of input variables. The RF algorithm has been demonstrated to have excellent performance in comparison to other machine learning algorithms [3], [16], [17].

1.2 Gini importance

The RF method has extremely useful byproducts, for instance, variable importance measures [3], [18]. There are two different algorithms for calculating variable importance.

The first algorithm is based on the Gini criterion used to create a classification tree, CART [12]. In this paper, we call the measure "Gini importance." At each node, decreases in Gini impurity are recorded for all variables used to form the split. Gini impurity $\Delta GI(t)$ is defined as follows:

$$\Delta GI(t) = P_t GI(t) - P_L GI(t_L) - P_R GI(t_R).$$

Here, $GI(t)$ is called the Gini index and is defined as follows:

$$GI(t) = 1 - \sum_k p(k \mid t)^2,$$

where $p(k \mid t)$ is the rate at which class k is discriminated correctly at node t, $GI(t_L)$ is a Gini index on the left side of the node, $GI(t_R)$ is a Gini index on the right side of the node, P_t is the number of samples before the split, P_L is the number of samples on the left side after the split, and P_R is the number of samples on the right side after the split. The Gini criterion is used to select the split with the highest impurity at each node. The average of all decreases in Gini impurity yields the "Gini importance" measure.

The second algorithm is based on OOB observations. In this paper, we call the measure "mean decrease accuracy." Although the structure of a decision tree provides information concerning important variables, such an interpretation is difficult for hundreds of trees in an ensemble. One additional feature of RF is the ability to evaluate the importance of each input variable by the OOB estimates. To evaluate the importance of each variable, the values of each variable in the OOB samples are allowed to permute. The perturbed OOB samples will run down each tree again. Then, the difference between the accuracies of the original and perturbed OOB samples over all trees in RF are averaged.

Variable importance of "mean decrease accuracy" is defined as follows: Let $X_j (j = 1, \cdots, M)$ be the permuted variables, where M is the number of all variables. X_j and the remaining nonpermuted predictor variables together form a perturbed OOB sample. When X_j is used to predict the response for the OOB sample, the prediction accuracy (i.e., the number of samples classified correctly) decreases

substantially, if the original variable X_j is associated with the response. For each tree f of the forest, consider the associated OOB_f sample (data not included in the bootstrap samples used to construct f). The error of a single tree f in this OOB_f sample is denoted by $errOOB_f$. Now, randomly permute the values of X_j in OOB_f to get a permuted sample denoted by OOB_{fj} and compute $errOOB_{fj}$, the error of predictor f in the perturbed sample. Variable importance of X_j is then equal to

$$VI(X_j) = \frac{1}{ntree} \sum_{f} (errOOB_f - errOOB_{f_j}),$$

where the summation is over all trees f of RF and $ntree$ denotes the number of trees of RF.

2. Rapid Feature Selection Based on Random Forests

We investigate the ranking of important variables derived from various datasets by using the original RF method and obtain an empirical rule: the rankings of important variables obtained from "Gini importance" and "mean decrease accuracy" differ slightly, whereas the members of the top ranked variables are almost the same. Thus, if we can determine these members of the top ranked variables obtained from "Gini importance," we can rank variable importance by "mean decrease accuracy."

To realize this idea, we combine "Gini importance" and "mean decrease accuracy" as "filter" and "wrapper." We propose an improved method of RF and call it "rapid feature selection" method (RFS). After reducing meaningless variables by "filter," rapid feature selection evaluates variable importance by "wrapper."

"Gini importance" can be acquired from the generation process of weak learners, thus it is convenient to use the "Gini importance" measure as a "filter." However, sometimes we cannot obtain high accuracy by only using such a "filter." On the other hand, "mean decrease accuracy" is high; "mean decrease accuracy" is computationally heavy because it has to call on the learning algorithm to evaluate each subset.

The rapid feature selection algorithm is as follows:

1) Exclude OOB data and draw $ntree$ bootstrap samples from training data.
2) For each bootstrap sample, randomly sample $mtry$ variables, grow a tree up to the first node, and record all Gini impurities generated in the calculation process.
3) Choose the top v variables that are candidates for the best split, give a score that reflects the top ranked v variables for $ntree$ trees, and aggregate all scores.
4) To select the top s important variables, choose the top s_e variables at this point (s_e is larger than s).
5) Rank s_e variables by "mean decrease accuracy" of the original RF method and select the top v variables.

Table 1: Information about each dataset

Dataset	Samples	Variables	Class	Accuracy
Internet Advertisements	3,279	1,558	2	0.963
Gisette	6,000	5,000	2	0.964

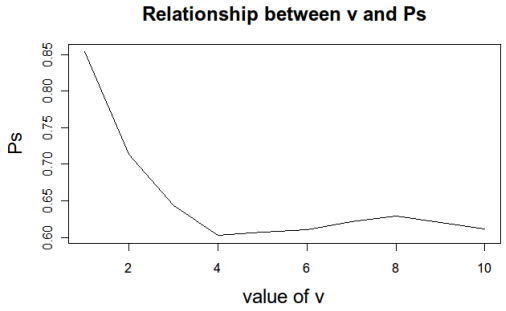


Fig. 1: Simulation: Relationship between v and P_s. Number of variables: $1,558$.

In the case that some variables are correlated, CART can choose the best split. However, CART needs the calculation of Gini impurity up to $2^{n-1} - 1$ times in the worst case, where n is the number of samples in each bootstrap sample. Thus, reducing the calculation time of CART is a significant issue in this method.

To reduce the calculation time of CART, some RF applications have an option to stop calculation at the first node. This option is effective in reducing computation time; however, the appropriate evaluation of important variables cannot be obtained. Necessary information will be insufficient when $v = 1$ owing to the nature of the data; therefore, we set a parameter v.

Under the assumption that CART can accurately rank variables and all variables are independent, we simulated the behavior of these parameters. In the simulation, we used the number of variables from Table 1.

Let P_s be a probability that the top s variables are included in the top s_e variables. The relationship among P_s and the other parameters are shown in Figures 1,2,3,4 and 5. The parameters that are not a target of the investigation are set as follows: $s_e = 35$, $s = 20$, $v = 5$ and $ntree = 100$ for the case of $1,558$ variables (Figures 1,3,4 and 5), and $s_e = 70$, $s = 55$ and $ntree = 100$ for the case of $5,000$ variables (Figure 2).

From Figures 1 and 2, we can find that the optimal v changes owing to the number of variables. Because CART cannot necessarily rank variables correctly and all variables are not independent in real data, in practice, the optimal v differs from the result of the simulation. Without changing the parameter setting, we conducted a experiment using real data to investigate about v. Internet advertisement dataset in Table 1 was chosen as a real data with $1,558$ variables. This

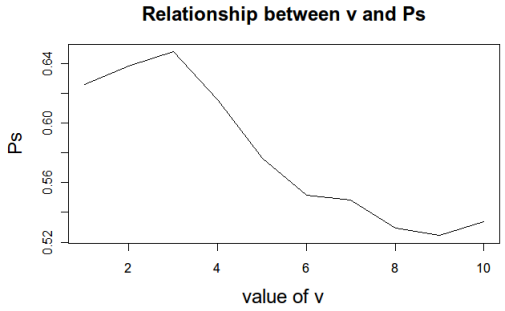


Fig. 2: Simulation: Relationship between v and P_s. Number of variables: $5,000$.

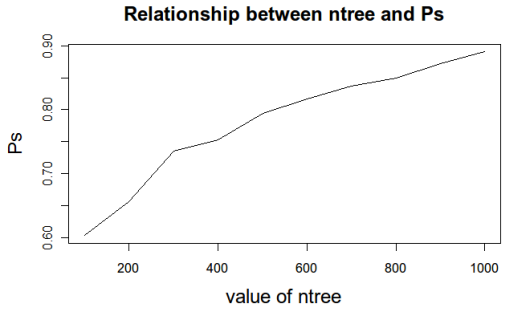


Fig. 4: Simulation: Relationship between $ntree$ and P_s. Number of variables: $1,558$.

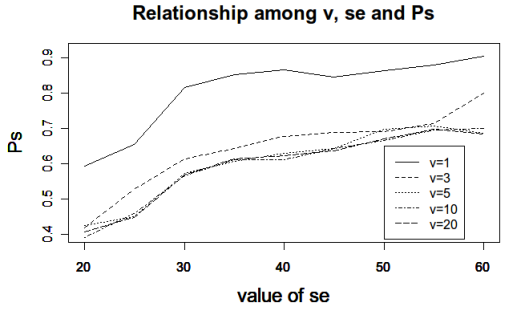


Fig. 3: Simulation: Relationship among v, s_e and P_s. Number of variables: $1,558$.

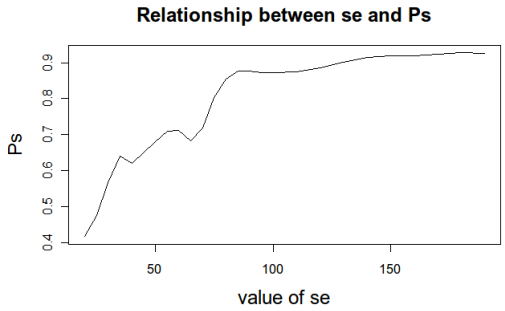


Fig. 5: Simulation: Relationship between s_e and P_s. Number of variables: $1,558$.

experiment was conducted using rapid feature selection.

Figure 6 shows that the accuracy of this real data is insensitive to the value of v. It is difficult to predict the optimal v. However, under the condition that v is 5 or more, we found that the behavior of P_s is stabilized if s_e changes. Figure 3 supports this empirical rule. Thus, we conducted experiments by provisionally setting $v = 5$, as described in the following chapter. Prediction of v is one of the future work.

Figure 4 expresses the relationship between $ntree$ and P_s, and Figure 5 the relationship between s_e and P_s. These Figures show a following relationship: The more the value of $ntree$ or s_e becomes large, the more the value of P_s approaches 1. Under the assumption that s is determined beforehand, we consider that all parameters should be set to satisfy the following condition:

$$mtry \times \frac{s}{M} \times P_s(s, s_e) \times ntree \geq s.$$

It is expected that the maximization of P_s and minimization of s_e and $ntree$ are realized simultaneously. When $1.5s = s_e, mtry \times ntree/M = 2.5$, P_s is about 0.5 is considered as one index.

3. Experiment

First, we conducted experiments to verify the performance of rapid feature selection compared with another well-known method. For comparison, we chose principal component analysis (PCA). PCA provides factor loading amount and accumulated contribution rate for variable selection. By using these values, we selected meaningful variables.

Next, to determine whether "mean decrease accuracy" used as "wrapper" in our method works effectively, we compared the performance of rapid feature selection and a method that employs only "filter" in rapid feature selection. In this paper, we refer to this method as "first split" (FS). First split does not use the evaluation from mean decrease accuracy. The first split algorithm is simple and its steps $1)$ to $3)$ are the same as those of the rapid feature selection algorithm, except that there is no need to exclude OOB data.

Because "mean decrease accuracy" consumes computation time, an alternative method is desired. To this end, we introduce weighted sampling. Gender et al. suggested selecting random $mtry$ inputs according to a distribution derived from the preliminary ranking given by a pilot estimator [19]. Based on their concept, we propose another method for rapid variable selection. In this paper, we call this method "first

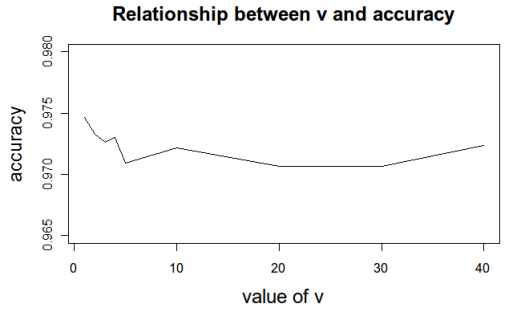


Fig. 6: Experiment: Relationship between v and $accuracy$. Dataset: Internet Advertisements.

split Gibbs" (FSG). After performing the first split algorithm, first split Gibbs normalizes the score derived from step 3) of the first split algorithm. Then, let the normalized values be $G_i (i = 1, \cdots, M)$ and calculate the Gibbs distribution by substituting G_i as a potential. The probability function of the Gibbs distribution is defined as follows:

$$P_i = \frac{\exp(-\beta G_i)}{\sum_{i=1}^{M} \exp(-\beta G_i)} \qquad (\beta > 0).$$

To sample $mtry$ variables according to the Gibbs distribution, first split Gibbs repeats the first split algorithm once again. Weighted samplings are performed by adjusting the parameter β. The original RF method samples $mtry$ variables according to the uniform distribution. Substituting $\beta = 0$ for the probability function of the Gibbs distribution, the resulting distribution equals the uniform distribution. When we substitute large values for β, the probability that the variables with large G_i are chosen increases.

Using high-dimensional data from UCI Machine Learning Repository, we investigate computation time and quality of variable selection, that is, whether important variables are properly selected. After performing PCA and the three methods, the accuracy of each is compared using only the variables selected. The score at step 3) of the rapid feature selection algorithm is obtained by giving the $1/r$ points to the rth variable ($r = 1, \cdots, v$).

As datasets for the experiment, we use an internet advertisements dataset and the Gisette dataset. Readers can refer to the details of these datasets at (http://archive.ics.uci.edu/ml/data-sets/Internet+Advertisements, http://archive.ics.uci.edu/ml/datasets/Gisette).

The experiment using internet advertisements results in trees with several nodes. On the other hand, the experiment using the Gisette dataset results in trees with many nodes. For each dataset, Table 1 shows the number of samples and variables and the accuracy calculated using all variables.

The computation environment is as follows: CPU Phenom X4 9950, OS Windows7 Professional 64bit, RAM 8GB.

Table 2: Comparison of computation time. (sec.)

Dataset	FS	FSG	RFS	RF	PCA
Internet Advertisements	8.39	16.21	9.22	272.46	38.50
Gisette	63.00	58.49	76.38	801.61	833.60

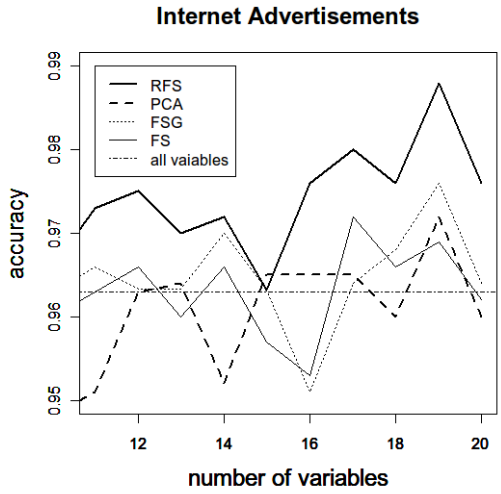


Fig. 7: Comparison of accuracy calculated using selected variables only. Method: RFS, PCA, FS and FSG. Dataset: Internet Advertisements.

4. Results and discussion

Table 2 shows the computation time of each method. The parameters used in this experiment are set as follows: $mtry = \lfloor \sqrt{M} + 0.5 \rfloor$, $ntree = 200$, $v = 5$, $s_e = 20$, $s = 15$. First split Gibbs and rapid feature selection need two-stage estimations. At each stage, $ntree = 100$ is set.

Computation time depends on the property of a dataset, thus the ranking of first split, first split Gibbs, and rapid feature selection varied slightly. However, the computation time of the rapid feature selection method was always lower than the original RF. From this result, rapid feature selection was found to be a much faster method than the original RF method.

The results of the accuracy calculated using selected variables only are plotted in Figures 7, 8 and 9. We can compare rapid feature selection, PCA, first split and first split Gibbs from these figures. In this experiment, the accuracy in Table 1 is used as the evaluation criterion regarding whether the information for classification is maintained. The result showed that rapid feature selection can maintain accuracy even if the number of dimensions becomes high.

The parameters used in this experiment are set as follows: $mtry = \lfloor \sqrt{M} + 0.5 \rfloor$, $ntree = 200$, $v = 5$, $\beta = 100$,$s = 10 - 20$, $s_e = 25 - 35$ for the internet advertisements dataset and $mtry = \lfloor \sqrt{M} + 0.5 \rfloor$, $ntree = 200$, $v = 5$, $\beta = 100$,$s = 45 - 55$, $s_e = 60 - 70$ for the Gisette dataset.

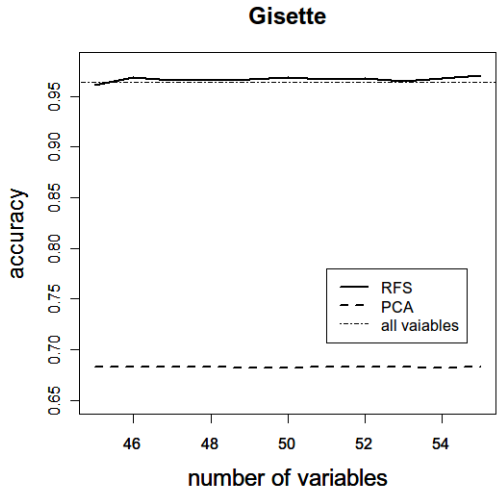


Fig. 8: Comparison of accuracy calculated using selected variables only. Method: RFS and PCA. Dataset: Gisette.

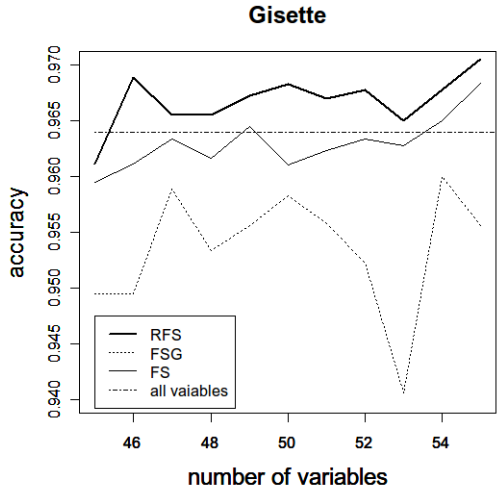


Fig. 9: Comparison of accuracy calculated using selected variables only. Method: RFS, FS and FSG. Dataset: Gisette.

Rapid feature selection needs two-stage estimations. At each stage, $ntree = 100$ is set.

From the results, we found that rapid feature selection can select important variables more accurately than first split and first split Gibbs. In addition, we found that trees with many nodes do not affect the results. Even if we reduced the number of variables to 0.6% for the internet advertisements dataset and 0.92% for the Gisette dataset, the accuracy did not fall below the evaluation criterion. These results indicate that rapid feature selection maintains the information for discrimination after variable selection.

The ranking of variables selected by each method are illustrated in Table 3. The results of the internet advertisements dataset are used for this experiment. The parameters used in this experiment are set as follows: $mtry = \lfloor \sqrt{M} + 0.5 \rfloor$, $ntree = 200$, $v = 5$, $\beta = 100$, $s_e = 34$, $s = 19$.

In the table, the values under the three methods represent the ID number of the variables. In this case, the ID number is up to $1,558$. Both rapid feature selection and first split select almost the same variables because rapid feature selection is based on first split. Here, about 11% of the variables are replaced, and the accuracy increased as a result of this change. Because "mean decrease accuracy" is introduced in step $5)$ of the rapid feature selection algorithm, the accuracy of rapid feature selection is higher than that of first split. Therefore, the effectiveness of the "wrapper" method was verified through this experiment.

First split Gibbs is also based on first split and about 37% of variables are replaced by weighted samplings. In this case, the estimation by sampling $mtry$ variables according to the Gibbs distribution was successful and accuracy was improved.

However, owing to the nature of the data, first split itself can correctly select important variables. In contrast, first split Gibbs reduces accuracy rate in such a situation. This phenomenon can be observed in Figure 9. Adjusting the value of β is difficult, thus first split Gibbs has a problem of time to adjust the value of β. However, first split Gibbs is a promising method as an alternative method of "mean decrease accuracy," if adjustment of β can be performed well.

Our study showed that rapid feature selection performs faster than the original RF method and can correctly select important variables even if trees with many nodes are generated. Rapid feature selection cannot search the minimum subset of significant variables for discrimination. However, under the conditions that the number of variables to be selected is predefined, rapid feature selection is useful to rapidly search essential variables.

5. Conclusion

In this paper, we proposed the rapid feature selection method based on an empirical rule: the rankings of important variables obtained from "Gini importance" and "mean decrease accuracy" differ slightly, whereas the members of the top ranked variables in RF are almost the same. If this empirical rule is solved mathematically, the reason our method is successful becomes clear.

The rapid feature selection method involves a two-step estimation. As the first step, candidates for important variables are chosen by a type of "filter." At this stage, variable importance is evaluated on the basis of "Gini importance." In the second stage, we select important variables by "wrapper." "Mean decrease accuracy" is adopted as the measure of variable importance. We calculate "mean decrease accuracy" using only variables chosen in the first stage. This is the reason rapid feature selection can maintain speed and accuracy.

Table 3: Illustration of variables selected by each method

Ranking	FS	FSG	RFS	PCA
1	3	3	352	2
2	1,425	1,154	1,400	1
3	1	2	1,484	3
4	2	352	3	1,244
5	969	1	1,425	1,484
6	1,154	1,400	1	1,456
7	1,423	1,484	2	1,436
8	1,199	969	1,154	352
9	1,556	1,119	1,423	1,400
10	1,255	347	1,199	1,279
11	1,119	458	1,556	549
12	1,345	896	1,255	918
13	1,400	994	1,119	360
14	1,484	1,048	1,345	541
15	1,214	1,109	1,555	557
16	1,555	1,199	1,048	337
17	352	1,225	1,109	915
18	1,048	1,230	1,144	173
19	1,109	1,424	820	1,363
Accuracy	0.973	0.982	0.979	0.973

The experimental results for computation time demonstrated that rapid feature selection is significantly faster than the original RF method. Although computation time depends on the nature of the data and the number of variables expected to be selected, it is certain that rapid feature selection selects important variables much faster than the original RF method when dealing with high-dimensional data.

Rapid feature selection was also found to be able to select important variables and maintain the information for classification. In the experiment, although the number of variables was reduced to about 0.8% and only 200 weak learners were used, rapid feature selection preserved a high degree of accuracy. These results show that our proposed method performance is sufficient for rapid variable selection.

By using rapid feature selection for various types of high-dimensional data, a means to improve the score generated at step $3)$ of the rapid feature selection algorithm may be found. Computation time may be further reduced by the combination of improved first split Gibbs and rapid feature selection. Moreover, it is necessary to not only collect empirical rules but also mathematical proof for the development of rapid feature selection.

References

[1] R. Kohavi and G. John, "Wrappers for feature subset selection," *Artificial intelligence*, vol. 97, no. 1-2, pp. 273–324, 1997.

[2] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *The Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[3] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[4] J. Chan and D. Paelinckx, "Evaluation of random forest and adaboost tree-based ensemble classification and spectral band selection for ecotope mapping using airborne hyperspectral imagery," *Remote Sensing of Environment*, vol. 112, no. 6, pp. 2999–3011, 2008.

[5] R. Díaz-Uriarte and S. De Andres, "Gene selection and classification of microarray data using random forest," *BMC bioinformatics*, vol. 7, no. 1, p. 3, 2006.

[6] P. Granitto, C. Furlanello, F. Biasioli, and F. Gasperi, "Recursive feature elimination with random forest for ptr-ms analysis of agroindustrial products," *Chemometrics and Intelligent Laboratory Systems*, vol. 83, no. 2, pp. 83–90, 2006.

[7] R. Genuer, J. Poggi, and C. Tuleau-Malot, "Variable selection using random forests," *Pattern Recognition Letters*, vol. 31, no. 14, pp. 2225–2236, 2010.

[8] C. Strobl, A. Boulesteix, T. Kneib, T. Augustin, and A. Zeileis, "Conditional variable importance for random forests," *BMC bioinformatics*, vol. 9, no. 1, p. 307, 2008.

[9] C. Strobl, A. Boulesteix, A. Zeileis, and T. Hothorn, "Bias in random forest variable importance measures: Illustrations, sources and a solution," *BMC bioinformatics*, vol. 8, no. 1, p. 25, 2007.

[10] K. Archer and R. Kimes, "Empirical characterization of random forest variable importance measures," *Computational Statistics & Data Analysis*, vol. 52, no. 4, pp. 2249–2260, 2008.

[11] B. Menze, B. Kelm, R. Masuch, U. Himmelreich, P. Bachert, W. Petrich, and F. Hamprecht, "A comparison of random forest and its gini importance with standard chemometric methods for the feature selection and classification of spectral data," *BMC bioinformatics*, vol. 10, no. 1, p. 213, 2009.

[12] L. Breiman, *Classification and regression trees*. Chapman & Hall/CRC, 1984.

[13] T. Ho, "The random subspace method for constructing decision forests," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 20, no. 8, pp. 832–844, 1998.

[14] M. Skurichina and R. Duin, "Bagging and the random subspace method for redundant feature spaces," *Multiple Classifier Systems*, pp. 1–10, 2001.

[15] R. Bryll, R. Gutierrez-Osuna, and F. Quek, "Attribute bagging: improving accuracy of classifier ensembles by using random feature subsets," *Pattern Recognition*, vol. 36, no. 6, pp. 1291–1302, 2003.

[16] V. Svetnik, A. Liaw, C. Tong, J. Culberson, R. Sheridan, and B. Feuston, "Random forest: a classification and regression tool for compound classification and qsar modeling," *Journal of chemical information and computer sciences*, vol. 43, no. 6, pp. 1947–1958, 2003.

[17] D. Meyer, F. Leisch, and K. Hornik, "The support vector machine under test," *Neurocomputing*, vol. 55, no. 1-2, pp. 169–186, 2003.

[18] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[19] R. Genuer, J. Poggi, and C. Tuleau, "Random forests: some methodological insights," *Arxiv preprint arXiv:0811.3619*, 2008.

Performance Evaluation of Some Inverse Iteration Algorithms on PowerXCell^{TM} 8i Processor

Masami Takata[1], **Hiroyuki Ishigami**[2], **Kinji Kimura**[2], **and Yoshimasa Nakamura**[2]
[1]Academic Group of Information and Computer Sciences, Nara Women's University, Nara, Nara, JAPAN
[2]Graduate School of Informatics, Kyoto University, Kyoto, Kyoto, JAPAN

Abstract— *In this paper, we compare with the inverse iteration algorithms on PowerXCell^{TM} 8i processor, which has been known as a heterogeneous environment. When some of all the eigenvalues are close together or there are clusters of eigenvalues, reorthogonalization must be adopted to all the eigenvectors associated with such eigenvalues. Reorthogonalization algorithms need a lot of computational cost. The Classical Gram-Schmidt (CGS) algorithm, the modified Gram-Schmidt (MGS) algorithm, and the Householder orthogonalization algorithm in terms of the compact WY representation have been known as reorthogonalization algorithms. These algorithms can be computed using BLAS level-1 and level-2. Since synergistic processor elements in PowerXCell^{TM} 8i processor archive the high performance of BLAS level-2 and level-3, the orthogonalization algorithms except the MGS algorithm can be computed high-speed on parallel computers.*

Keywords: inverse iteration, eigenvalue decomposition, Classical Gram-Schmidt, Householder transformation, modified Gram-Schmidt, PowerXCell^{TM} 8i processor

1. Introduction

The eigenvalue decomposition of a symmetric matrix is one of the most important operations in linear algebra. It is used in molecular orbital of chemical, vibrational analysis, image processing, data searches, etc..

Owing to recent improvements in the performance of computers equipped with multicore processors, we have had more opportunities to perform calculations on parallel computers. As a result, there has been an increase in the demand for an eigenvalue decomposition algorithm that can be effectively parallelized.

Any $n \times n$ symmetric matrix is transformed into a symmetric tridiagonal matrix by using a sequence of Householder transformations [4], [9]. This preconditioning process helps to shorten computational time drastically. Hence, eigenvalue decomposition algorithms of symmetric tridiagonal matrices are important. Several eigenvalue decomposition algorithms of a symmetric tridiagonal matrix have been proposed [3], [7], [10], [12], [13], [17]. They are classified into two types. The first type of algorithm computes simultaneously all the eigenvalues and the eigenvectors. Algorithms of this type include the QR algorithm [10] and the divide-and-conquer algorithm [3], [13]. The second type of algorithm computes all or some eigenvalues and all or some eigenvectors. Algorithms for computing eigenvalues include the root-free QR algorithm [12] and the bisection algorithm [10]. Algorithms for computing eigenvectors include the MR^3 algorithm [7] and the inverse iteration algorithm with the modified Gram-Schmidt (MGS) algorithm [10], [17]. LAPACK (Linear Algebra PACKage) [16], which is a software library for numerical linear algebra, has codes that integrate all the above-mentioned algorithms. These algorithms can be parallelized, except the root-free QR algorithm.

The inverse iteration algorithm is an algorithm for computing eigenvectors independently associated with mutually distinct eigenvalues. However, when some eigenvalues are very close to each other, the eigenvectors, which are computed using the inverse iteration algorithm, must be reorthogonalized. As reorthogonalization algorithms, the Classical Gram-Schmidt (CGS) algorithm [10], the MGS algorithm, the Householder orthogonalization algorithm [15] are known. Reorthogonalization algorithms need a lot of computational cost. The CGS algorithm is suitable algorithm for parallel computing. The orthogonality of eigenvectors computed by the CGS algorithm depends on the square of the condition number of the eigenvectors, which are generated using the inverse iteration, in the same cluster of the eigenvalues [20]. The MGS algorithm is sequential and inefficient for parallel computing. The orthogonality of eigenvectors computed by the MGS algorithm depends on the condition number. The Householder orthogonalization algorithm can orthogonalize eigenvectors by using the Householder transformation [19]. The orthogonality in the Householder orthogonalization algorithm does not depend on the condition number. The Householder algorithm is sequential and inefficient for parallel computing. Ishigami et. al. have developed parallel algorithms for the Householder orthogonalization algorithm in terms of the compact WY representation [15], which is named as the cWY algorithm in this paper.

In ExaFLOP computing, since it is critical issue to minimize electricity, heterogeneous environments are suitable. Consequently, it is important to validate the inverse iteration algorithms with the CGS algorithm, the MGS algorithm, and the cWY in heterogeneous environments. As a heteroge-

neous environment, cell processor has PowerPC Processor Element (PPE) and eight cores of Synergistic Processor Elements (SPEs). PPE and SPEs can share the same memory. Since SPEs are consisted as multicore, SPEs archive the high performance of BLAS level-2 and level-3 [1]. Basic Linear Algebra Subprograms (BLAS) is an application programming interface standard for publishing libraries to perform basic linear algebra operations such as vector and matrix multiplications. BLAS level-1 can compute vector operations such as inner products, dot products and vector norms. BLAS level-2 and level-3 contain matrix-vector and matrix-matrix operations, respectively. The CGS algorithm and the MGS algorithm can be computed using BLAS level-2 and level-1, respectively. The cWY needs BLAS level-1 and level-2. Note that, the Householder orthogonalization algorithm is almost computed using BLAS level-2. Therefore, these orthogonalization algorithms should be performed in SPEs. By using PPE, an implementation of an inverse iteration is easy. In this paper, we compare with the CGS algorithm, the MGS algorithm, and the cWY on PowerXCell^{TM} 8i processor.

In Section 2, we give a brief review on eigenvalue decomposition. In Section 3, we explain an inverse iteration algorithm and describe its orthogonalization algorithms. In Section 4, we confirm each performance in the inverse iteration algorithms with orthogonalization algorithms on PowerXCell^{TM} 8i processor.

2. Eigenvalue decomposition

Let A be $n \times n$ matrix such that

$$A\boldsymbol{v}_j = \lambda_j \boldsymbol{v}_j \quad (j = 1, 2, ..., n) \tag{1}$$

where λ_j ($\lambda_j : \lambda_j \in \mathbb{C}$) and $\boldsymbol{v}_j$ ($\boldsymbol{v}_j : \boldsymbol{v}_j (\neq 0) \in \mathbb{C}^n$) are an eigenvalue and an eigenvector of A, respectively. If eigenvectors $\boldsymbol{v}_j$ of A are linear independent, then

$$AV = VD, \tag{2}$$

$$D = diag\left[\lambda_1 \quad \lambda_2 \quad \cdots \quad \lambda_n\right], \tag{3}$$

$$V = \left[\boldsymbol{v}_1 \quad \boldsymbol{v}_2 \quad \cdots \quad \boldsymbol{v}_n\right]. \tag{4}$$

Since V is nonsingular, the inverse matrix V^{-1} exists and $V^{-1}V$ is equal to an identity matrix I. Hence, A is decomposed as

$$A = VDV^{-1} \tag{5}$$

Eq.(5) is called eigenvalue decomposition of A.

Let A be real symmetric, then $\lambda_j \in \mathbb{R}$ and $\boldsymbol{v}_j \in \mathbb{R}$. Moreover, eigenvectors $\boldsymbol{v}_j$ are orthogonal to each other, if $\lambda_1 \neq \lambda_2 \neq \cdots \neq \lambda_n$. Note here that V becomes orthogonal matrix by the normalization $\boldsymbol{v}_j \rightarrow \boldsymbol{v}_j / \|\boldsymbol{v}_j\|$. Then A is decomposed as

$$A = VDV^{\top} \tag{6}$$

where $V^{\top}$ denotes the transposed matrix of V.

In a famous algorithm, a real symmetric matrix A is similarly transformed into a symmetric tridiagonal matrix T by using the Householder transformations. Namely,

$$Q_A^{\top} A Q_A = T, \tag{7}$$

with suitable orthogonal matrix Q_A. After the tridiagonalization, T is decomposed as

$$T = Q_T D Q_T^{\top} \tag{8}$$

by some orthogonal matrix Q_T. Consequently, by combining Eq.(7) with Eq.(8), the eigenvalue decomposition of A is given as

$$A = (Q_A Q_T) D (Q_A Q_T)^{\top}. \tag{9}$$

3. Inverse iteration algorithm

In this section, we introduce the inverse iteration algorithm. When some of all the eigenvalues are close together or there are clusters of eigenvalues, reorthogonalization must be needed to all the eigenvectors associated with such eigenvalues, since the eigenvectors needs to be orthogonal to each other. Therefore, reorthogonalization algorithms should be adopted.

In Section 3.1, we explain a concept of the inverse iteration algorithm. In Section 3.2, 3.3, and 3.4, the CGS algorithm, the MGS algorithm and the cWY are described, respectively. In Section 3.5, these orthogonalization algorithm are compared. In Section 3.6, we descrive a relationship between BLAS and the orthogonalization algorithms.

3.1 Concept

When $\tilde{\lambda_j}$ is an approximate value of λ_j and a starting vector $\boldsymbol{v}_j^{(0)}$ are given, the inverse iteration algorithm can compute an eigenvector of T. To this end, the following equation is solved iteratively:

$$\left(T - \tilde{\lambda_j} I\right) \boldsymbol{v}_j^{(k)} = \boldsymbol{v}_j^{(k-1)} \tag{10}$$

If the eigenvalues of T are mutually well-separated, the solution of $\boldsymbol{v}_j^{(k)}$ in Eq.(10) generically converges to the eigenvector associated with λ_j as k goes to ∞ The above iteration algorithm is the inverse iteration algorithm. When m eigenvectors are computed, the computational cost of this algorithm is of order mn. The computational cost is less than that of other algorithms.In the implementation, the vector $\boldsymbol{v}_j^{(k)}$ must be normalized to avoid overflow.

3.2 Classical Gram-Schmidt algorithm

The CGS algorithm has been proposed as the first reorthogonalization algorithm. In the CGS algorithm, a basis

1: $\boldsymbol{x}_1 = \boldsymbol{v}_1$.
2: **for** $j = 2$ to m **do**
3: Generate $\boldsymbol{v}_j$ in an algorithm.
4: Eq.(11) and Eq.(12) : Orthogonalize $\boldsymbol{v}_j$ to $\boldsymbol{x}_j$ by using $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_{j-1}$.
5: **end for**

Fig. 1: Classical Gram-Schmidt algorithm.

1: **for** $j = 1$ to n **do**
2: Generate $\boldsymbol{v}_j^{(0)}$ from random numbers.
3: $k = 0$
4: **repeat**
5: $k \leftarrow k + 1$.
6: Normalize $\boldsymbol{v}_j^{(k-1)}$.
7: Eq.(10) : Compute $\boldsymbol{v}_j^{(k)}$ by using $\boldsymbol{v}_j^{(k-1)}$.
8: **if** $|\tilde{\lambda_j} - \tilde{\lambda}_{j-1}| \leq 10^{-3}\|T\|$, **then**
9: **for** $i = j_1$ to $j - 1$ **do**
10: $\boldsymbol{v}_j^{(k)} \leftarrow \boldsymbol{v}_j^{(k)} - [\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_{j-1}] \begin{bmatrix} \boldsymbol{x}_1^\top \\ \boldsymbol{x}_2^\top \\ \vdots \\ \boldsymbol{x}_{j-1}^\top \end{bmatrix} \boldsymbol{v}_j^{(k)}$
11: **end for**
12: **else**
13: $j_1 = j$
14: **end if**
15: **until** some condition is met.
16: Normalize $\boldsymbol{v}_j^{(k)}$ to $\boldsymbol{x}_j$.
17: **end for**

Fig. 2: Inverse iteration algorithm with the CGS algorithm. j_1 means the index j of the first eigenvalue of a cluster.

vector $\boldsymbol{x}_j$, which is an orthogonal vector in $\boldsymbol{v}_j$, is computed as follows:

$$\boldsymbol{x}'_j = \boldsymbol{v}_j - \sum_{i=1}^{j-1} \langle \boldsymbol{v}_j, \boldsymbol{x}_i \rangle \boldsymbol{x}_i, \tag{11}$$

$$\boldsymbol{x}_j = \frac{\boldsymbol{x}'_j}{\|\boldsymbol{x}'_j\|} \tag{12}$$

In Eq.(11), $\langle \boldsymbol{v}_j, \boldsymbol{x}_i \rangle \boldsymbol{x}_i$ means an orthographic projection on the direction to $\boldsymbol{x}_i$ of $\boldsymbol{v}_j$. Through $\boldsymbol{v}_j$ is subtracted the orthographic projection, $\boldsymbol{v}_j$ can be picked out of elements $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_{j-1}$. Thus, $\boldsymbol{x}_j$ is orthogonalized.

Figure 1 shows the orthogonalization algorithm using the CGS algorithm. Since Eq.(11) and Eq.(12) are computed using an inner product, BLAS level-1 has to be adopted. Therefore, to adopt BLAS level-2, Eq.(11) and Eq.(12) should be transformed into the following vector product.

$$\boldsymbol{x}'_j = \boldsymbol{v}_j - [\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_{j-1}] \begin{bmatrix} \boldsymbol{x}_1^\top \\ \boldsymbol{x}_2^\top \\ \vdots \\ \boldsymbol{x}_{j-1}^\top \end{bmatrix} \boldsymbol{v}_j. \tag{13}$$

Figure 2 is a code, which is based on DSTEIN in LAPACK and modified the orthogonalization process from the MGS algorithm to the CGS algorithm. Specifically, line 10 in Figure 2 is changed to Eq.(13).

1: **for** $j = 1$ to n **do**
2: Generate $\boldsymbol{v}_j^{(0)}$ from random numbers.
3: $k = 0$
4: **repeat**
5: $k \leftarrow k + 1$.
6: Normalize $\boldsymbol{v}_j^{(k-1)}$.
7: Eq.(10) : Compute $\boldsymbol{v}_j^{(k)}$ by using $\boldsymbol{v}_j^{(k-1)}$.
8: **if** $|\tilde{\lambda_j} - \tilde{\lambda}_{j-1}| \leq 10^{-3}\|T\|$, **then**
9: **for** $i = j_1$ to $j - 1$ **do**
10: $\boldsymbol{v}_j^{(k)} \leftarrow \boldsymbol{v}_j^{(k)} - \langle \boldsymbol{v}_j^{(k)}, \boldsymbol{x}_i \rangle \boldsymbol{x}_i$
11: **end for**
12: **else**
13: $j_1 = j$
14: **end if**
15: **until** some condition is met.
16: Normalize $\boldsymbol{v}_j^{(k)}$ to $\boldsymbol{x}_j$.
17: **end for**

Fig. 3: Inverse iteration algorithm with the MGS algorithm.

3.3 Modified Gram-Schmidt algorithm

If the MGS algorithm is adopted to reorthogonalize eigenvectors, the computational cost is of order $m^2 n$. Therefore, the computational cost, for which eigenvectors of a matrix T are computed, increases significantly. In general, to implement the inverse iteration algorithm on computers, the MGS algorithm with the Peters-Wilkinson method [17] is adopted as the standard orthogonalization process. The MGS algorithm with the Peters-Wilkinson method is also available on DSTEIN, which is implemented in the LAPACK code [16] of the inverse iteration algorithm for computing eigenvectors of a real symmetric tridiagonal matrix. In the Peters-Wilkinson method, when the distance between the close eigenvalues is less than $10^{-3}\|T\|$, these close eigenvalues are regarded as members of the same cluster of eigenvalues, and all of the eigenvectors associated with these eigenvalues are orthogonalized.

Figure 3 shows the inverse iteration algorithm based on the MGS algorithm with the Peters-Wilkinson method. This loop includes the iteration based on Eq.(10) and the orthogonalization of the eigenvectors. This orthogonalization process becomes a bottleneck of the inverse iteration with respect to the computational time. The MGS algorithm is mainly based on BLAS level-1 such as the inner product operation and the AXPY operation [1].

3.4 Householder orthogonalization algorithm

The Householder orthogonalization algorithm is one of the alternative orthogonalization algorithms. When some vectors $\boldsymbol{v}_j$, $\boldsymbol{w}_j \in \mathbb{R}^n$ satisfy $\|\boldsymbol{v}_j\|_2 = \|\boldsymbol{w}_j\|_2$, there exists the symmetric matrix H_j satisfying $H_j H_j^\top = H_j^\top H_j = I$, $H_j \boldsymbol{v}_j = \boldsymbol{w}_j$ defined by

$$H_j = I - s_j \boldsymbol{y}_j \boldsymbol{y}_j^\top, \tag{14}$$

where $\boldsymbol{y}_j = \boldsymbol{v}_j - \boldsymbol{w}_j$ and $s_j = 2/\|\boldsymbol{y}_j\|_2^2$. The transformation by H_j is called the Householder transformation. Figure 4 shows the Householder orthogonalization algorithm. The

1: **for** $j = 1$ to m **do**
2: Generate $\boldsymbol{v}_j$ in an algorithm.
3: $\boldsymbol{v}'_j = \left(I - s_{j-1}\boldsymbol{y}_{j-1}\boldsymbol{y}_{j-1}^\top\right)\cdots\left(I - s_2\boldsymbol{y}_2\boldsymbol{y}_2^\top\right)\left(I - s_1\boldsymbol{y}_1\boldsymbol{y}_1^\top\right)\boldsymbol{v}_j$.
4: Compute $\boldsymbol{y}_j$ and s_j by using $\boldsymbol{v}'_j$.
5: $\boldsymbol{x}_j = \left(I - s_1\boldsymbol{y}_1\boldsymbol{y}_1^\top\right)\left(I - s_2\boldsymbol{y}_2\boldsymbol{y}_2^\top\right)\cdots\left(I - s_j\boldsymbol{y}_j\boldsymbol{y}_j^\top\right)\boldsymbol{e}_j$.
6: **end for**

Fig. 4: Householder orthogonalization algorithm.

vector $\boldsymbol{y}_j$ is the vector, in which the elements from 1 to $j-1$ are the same as the elements of $\boldsymbol{v}'_j$ and the elements from $j+1$ to n are zero. $\boldsymbol{v}'_j$ and $\boldsymbol{w}_j$ are defined as follows:

$$\begin{aligned}\boldsymbol{v}'_j &= \begin{bmatrix} v'_{j\{1\}} & \cdots & v'_{j\{j-1\}} & v'_{j\{j\}} & \cdots & v'_{j\{n\}}\end{bmatrix}^\top \\ &= H_{j-1}H_{j-2}\cdots H_2H_1\boldsymbol{v}_j, \qquad (15)\\ \boldsymbol{w}_j &= \begin{bmatrix} v'_{j\{1\}} & \cdots & v'_{j\{j-1\}} & c_j & \boldsymbol{0}\end{bmatrix}^\top, \qquad (16)\end{aligned}$$

where,

$$c_j = -sgn\left(v'_{j\{j\}}\right)\sqrt{\sum_{i=j}^{n} {v'_{j\{i\}}}^2}. \qquad (17)$$

H_j, y_j and s_j are computed using $\boldsymbol{v}_j$ as follows:

$$\begin{aligned}H_j &= I - s_j\boldsymbol{y}_j\boldsymbol{y}_j^\top \qquad (18)\\ \boldsymbol{y}_j &= \boldsymbol{v}'_j - \boldsymbol{w}_j \qquad (19)\\ \|\boldsymbol{y}_j\|_2^2 &= (v'_{j\{j\}} - c_j)^2 + \sum_{i=j+1}^{n} {v'_{j\{i\}}}^2 \qquad (20)\\ &= \sum_{i=j}^{n} {v'_{j\{i\}}}^2 - 2v'_{j\{j\}}c_j + c_j^2 \qquad (21)\\ &= 2\left(c_j^2 - v'_{j\{j\}}c_j\right). \qquad (22)\\ s_j &= \frac{2}{\|\boldsymbol{y}_j\|_2^2} = \frac{1}{c_j^2 - v'_{j\{j\}}c_j}. \qquad (23)\end{aligned}$$

The vector $\boldsymbol{e}_j$ in Figure 4 is the j-th vector of an n-dimensional identity matrix.

The orthogonality of the vectors $\boldsymbol{x}_j$ generated by the Householder orthogonalization algorithm does not depend on the condition number of a matrix T. Therefore, the Householder orthogonalization algorithm is more stable than the MGS algorithm. On the other hand, being similar to the MGS algorithm, it is sequential algorithm that is mainly based on BLAS level-1. Its computational cost is higher than that of the MGS algorithm. Thus the Householder orthogonalization algorithm is an ineffective algorithm in parallel computing.

By combination with the compact WY representation [18], the Householder orthogonalization algorithm becomes capable of computation with BLAS level-2 [20]. Hence, in this paper, the cWY is adopted to an inverse iteration. Let $Y_1 = \boldsymbol{y}_1 \in \mathbb{R}^{n\times j}$ and $S_1 = s_1 \in \mathbb{R}^{1\times 1}$. Matrices Y_j

1: **for** $j = 1$ to m **do**
2: Generate $\boldsymbol{v}_j$ in an algorithm
3: $\boldsymbol{v}'_j = \left(I - Y_{j-1}S_{j-1}^\top Y_{j-1}^\top\right)\boldsymbol{v}_j$.
4: Compute $\boldsymbol{y}_j$ and s_j by using $\boldsymbol{v}'_j$.
5: Eq.(24) and Eq.(25) : Update Y_j and S_j by using s_j, $\boldsymbol{y}_j$, S_{j-1} and Y_{j-1}.
6: $\boldsymbol{q}_j = \left(I - Y_jS_jY_j^\top\right)\boldsymbol{e}_j$.
7: **end for**

Fig. 5: Householder orthogonalization algorithm in terms of the compact WY representation.

1: **for** $j = 1$ to n **do**
2: Generate $\boldsymbol{v}_j^{(0)}$ from random numbers.
3: $k = 0$
4: **repeat**
5: $k \leftarrow k + 1$.
6: Normalize $\boldsymbol{v}_j^{(k-1)}$.
7: Solve linear equations : $\left(T - \tilde{\lambda_j}I\right)\boldsymbol{v}_j^{(k)} = \boldsymbol{v}_j^{(k-1)}$.
8: **if** $|\tilde{\lambda_j} - \tilde{\lambda}_{j-1}| \leq 10^{-3}\|T\|$, **then**
9: $j_c \leftarrow j - j_1$.
10: **if** $j_c = 1$ and $k = 1$, **then**
11: Compute $Y_1 = \boldsymbol{y}_1$ and $S_1 = s_1$ by using $\boldsymbol{v}_{j_1}$.
12: **end if**
13: $\boldsymbol{v}'_{j_c+1} = \left(I - Y_{j_c}S_{j_c}^\top Y_{j_c}^\top\right)\boldsymbol{v}_j^{(k)}$.
14: Compute $\boldsymbol{y}_{j_c+1}$ and s_{j_c+1} by using $\boldsymbol{v}'_{j_c+1}$.
15: Eq.(24) and Eq.(25) : Update Y_{j_c+1} and S_{j_c+1} by using s_{j_c+1}, $\boldsymbol{y}_{j_c+1}$, S_{j_c} and Y_{j_c}.
16: $\boldsymbol{v}_j^{(k)} \leftarrow \left(I - Y_{j_c+1}S_{j_c+1}Y_{\mathrm{j_c}+1}^\top\right)\boldsymbol{e}_{\mathrm{j_c}+1}$.
17: **else**
18: $j_1 \leftarrow j$.
19: **end if**
20: **until** some condition is met.
21: Normalize $\boldsymbol{v}_j^{(k)}$ to $\boldsymbol{v}_j$.
22: **end for**

Fig. 6: Inverse iteration algorithm with the cWY algorithm.

and upper triangular matrices S_j is defined recursively as follows:

$$\begin{aligned}Y_j &= \begin{bmatrix} Y_{j-1} & \boldsymbol{y}_j\end{bmatrix}, \qquad (24)\\ S_j &= \begin{bmatrix} S_{j-1} & -s_jS_{j-1}Y_{j-1}^\top\boldsymbol{y}_j \\ \boldsymbol{0} & s_j\end{bmatrix}. \qquad (25)\end{aligned}$$

In this case, the following equation holds

$$H_1H_2\cdots H_j = I - Y_jS_jY_j^\top. \qquad (26)$$

As shown by Eq.(26), the product of the Householder matrices $H_1H_2\cdots H_j$ can be rewriten in a simple block matrix form. Here $I - Y_jS_jY_j^\top$ is called the compact WY representation of the product of the Householder matrices. Figure 5 shows the orthogonalization algorithm.

Figure 6 is a code, which is based on DSTEIN in LAPACK and changed the orthogonalization process from the MGS algorithm to the cWY algorithm. In other words, the MGS algorithm (from line 4 to 15 in Figure 3) is rewriten the cWY algorithm.In Figure 6, the index j_c denotes the j_c-th eigenvalue of the cluster in computing the j_c-th eigenvector. This index j_c needs to compute and update S_j and Y_j.

Table 1: Comparison of the orthogonalization algorithms [5] [20].

algorithms	*Computation*	*Synchronization*	*Orthogonality*
CGS	almost $2m^2n$	$O(m)$	$O(\epsilon\kappa(A)^2)$
MGS	almost $2m^2n$	$O(m^2)$	$O(\epsilon\kappa(A))$
House	almost $4m^2n$	$O(m^2)$	$O(\epsilon)$
cWY	almost $4m^2n$	$O(m)$	$O(\epsilon)$

Therefore, a variable j_c should be confirmed on line 9 in Figure 6.

The cWY algorithm has a stable orthogonality arising from the Householder transformations, and its mathematical calculation is mainly performed by BLAS level-2 such as the product of a matrix and a vector and a rank-1 update operation.

3.5 Comparison of the orthogonalization algorithms

The cWY algorithm has a stable orthogonality arising from the Householder transformations, and its mathematical calculation is mainly performed by BLAS level-2 such as the product of a matrix and a vector and a rank-1 update operation. As a result, this orthogonalization has more stable and sophisticated orthogonality, and it is more effective for parallel computing than the MGS algorithm. Table 1 displays the differences in performance of the four orthogonalization methods, considered in the above sections. In this table, "House" denotes the Householder orthogonalization algorithm. *Computation* denotes the order of the computational cost. *Synchronization* denotes the order of the number of synchronizations. *Orthogonality* denotes the norm $\|V^\top V - I\|$, where $V = [\boldsymbol{v}_1, \cdots, \boldsymbol{v}_n]$. ϵ denotes the machine epsilon and κ denotes the condition number of a matrix. These are the results obtained from [5] and [20].

On the other hand, the computational cost in the CGS algorithm is twice less than that in the cWY algorithm. Therefore, when high orthogonality is not needed, the CGS algorithm is also the suitable selection for the orthogonalization.

3.6 Adoption of BLAS

The line from 1 to 7 on each algorithm is the code in the inverse iteration algorithm without an orthogonalization algorithm. This computational cost mn is relatively smaller than that in the inverse iteration algorithm with an orthogonalization algorithm shown in Table 1. Therefore, we adopt SPEs to orthogonalization algorithms.

In the CGS algorithm, the line 10 on Figure 2 can be computed using BLAS level-2. In the MGS algorithm, BLAS level-1 is adopted in the line 10 on Figure 3. In the cWY algorithm, the line 13 and 16 on Figure 6 can be performed with BLAS level-2, and the line 11 and 14 can be performed with BLAS level-1.

4. Experiments

In this section, we describe some numerical experiments performed using the CGS algorithm, the MGS algorithm, and the cWY algorithm on PowerXCell^{TM} 8i processor.

In the experiments, we use GigaAccel 180, which is a PCI Express board with PowerXCell^{TM} 8i processor. PowerXCell^{TM} 8i processor is one of Cell Broadband Engine^{TM}. The theoretical performances of a single and double precision floating-point arithmetic operation on an SPE in PowerXCell^{TM} 8i processor are 180GFLOPS and 90GFLOPS in 2.8GHz, respectively. We implement those algorithms by using Cell SDK 3.1 [2], which is developed by the IBM corporate [14]. Cell SDK 3.1 includes the parallelized BLAS for Cell Broadband Engine^{TM}. The MGS algorithm is implemented in Cell SDK 3.1.

As experimental matrices, we use three types. Type 1 is a random matrix, of which elements are set to the random number on the interval from 0 to 1. Type 2 is shown as follows:

$$\begin{bmatrix} 1 & 1 & & & \\ 1 & 1 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 1 & 1 \\ & & & 1 & 1 \end{bmatrix}. \tag{27}$$

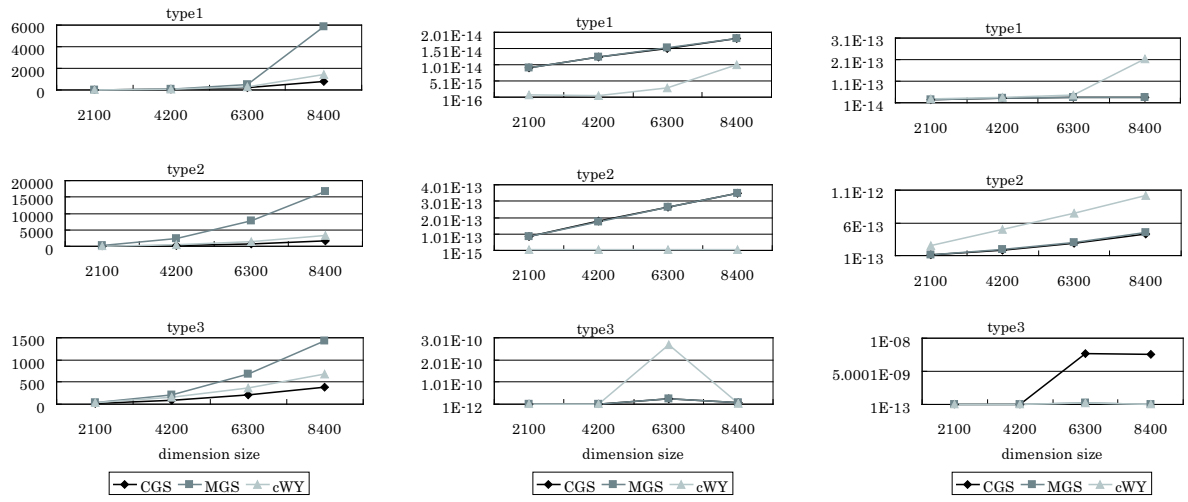
Type 3 is the glued-Wilkinson matrix $W_g^\dagger$, which is real symmetric and has dimensions on the order of thousands. The glued-Wilkinson matrix has been used to evaluate the performance of the inverse iteration algorithms as the benchmark problems of eigenvalue decomposition [6], [8]. $W_g^\dagger$ consists of the block matrix $W_{21}^\dagger \in \mathbb{R}^{21\times 21}$ and the scalar parameter $\delta \in \mathbb{R}^{1\times 1}$ and is defined as follow:

$$W_g^\dagger = \left[\begin{array}{c|c|cc|c} W_{21}^\dagger & \delta & & & \\ \hline \delta & W_{21}^\dagger & \delta & & \\ \hline & \delta & \ddots & \ddots & \\ & & \ddots & \ddots & \delta \\ \hline & & & \delta & W_{21}^\dagger \end{array}\right], \tag{28}$$

where $W_{21}^\dagger$ is defined by

$$W_{21}^\dagger = \begin{bmatrix} 10 & 1 & & & & \\ 1 & 9 & 1 & & & \\ & 1 & \ddots & \ddots & & \\ & & \ddots & 0 & \ddots & \\ & & & \ddots & \ddots & 1 \\ & & & & 1 & 10 \end{bmatrix}, \tag{29}$$

and δ satisfies $0 < \delta < 1$ and is also the semi-diagonal element of $W_g^\dagger$. Since $W_g^\dagger$ is real symmetric tridiagonal and its semi-diagonal elements are nonzero, all the eigenvalues of



(a) Computational time [sec.] (b) $\|AV - VD\|_F$ (c) $\|V^\top V - I\|_F$

Fig. 7: Relationship between dimension size and performance in the orthogonalization algorithms.

$W_g^\dagger$ are distinct and real, and they are divided into 21 clusters of close eigenvalues. When δ is small, the distance between the minimum and maximum eigenvalues in any cluster is small. In our experiments, we set $\delta = 10^{-4}$.

Figure 7 and Table 2 show the experimental results of the orthogonalization algorithms. Time in Table 2 is the computational time. $\|AV - VD\|_F$ and $\|V^\top V - I\|_F$ mean the frobenius norm of synchronization and orthogonalization, respectively.

In type 1, each eigenvalue is usually separated. On the other hand, eigenvalues in type 2 and 3 become cluster. Therefore, $\|AV - VD\|_F$ and $\|V^\top V - I\|_F$ are smaller than that in type 2 and 3.

In 2100 dimension size of type 1 in Table 2, the computational time in the MGS algorithm, which is implemented by the IBM corporate, is smaller than that in the other orthogonalization algorithms. However, the increasing rate of the computational time in the MGS algorithm is higher as shown in Figure 7(a). The MGS algorithm is computed using BLAS level-1. On the other hand, the CGS algorithm and the cWY algorithm are almost computed using BLAS level-2. Therefore, in the computational time, the CGS algorithm and the cWY algorithm are better.

In Figure 7(b) and Table 2, $\|AV - VD\|_F$ in the CGS algorithm is nearly equal to that in the MGS algorithm. $\|AV - VD\|_F$ of the cWY algorithm is the smallest, except the case of 6300 dimension size in type 3. The exception is likely to be caused by the order of $\boldsymbol{v}_j$. In the experiments, $\boldsymbol{v}_j$ is listed in descending order of eigenvalues, which are related to eigenvectors. Therefore, by using the cWY algorithm with suitable order of $\boldsymbol{v}_j$, accuracy of eigenvector computation can become more properly.

In type 1 and 2 of Figure 7(c), $\|V^\top V - I\|_F$ in the CGS algorithm is nearly equal to that in the MGS algorithm. The CGS algorithm and the MGS algorithm are focused on the orthogonality of eigenvectors. On the other hand, in the cWY algorithm, accuracy of eigenvalue decomposition is given importance. Therefore, the orthogonality of eigenvectors is something lower than that in the CGS algorithm and the MGS algorithm.

In type 3 of Figure 7(c), $\|V^\top V - I\|_F$ in the CGS algorithm is worse than that in the other orthogonalization algorithm. In $\delta = 10^{-4}$, eigenvalues in type 3 are extremely close together. Therefore, the CGS algorithm is aborted that $\boldsymbol{v}_j$ is picked out.

In summarization, the computational time in the cWY algorithm is adequate speedy. Furthermore, $\|AV - VD\|_F$ and $\|V^\top V - I\|_F$ in the cWY algorithm is sufficient accuracy. Hence, the cWY algorithm is suitable, except case that the high-orthogonality of eigenvectors is given importance.

5. Conclusions

In this paper, we validated the parallel performance of the inverse iteration algorithms with the CGS algorithm, the MGS algorithm, and the cWY algorithm on PowerXCell^{TM} 8i processor. PowerXCell^{TM} 8i processor is one of heterogeneous environments. In ExaFLOP computing, since it is critical issue to minimize electricity, heterogeneous environments are suitable. SPEs in PowerXCell^{TM} 8i processor archive the high performance of BLAS level-2 and level-3. The inverse iteration algorithms are algorithms for computing eigenvectors and need a lot of computational cost. Therefore, the algorithms should be computed with SPEs. The experimental results show that the computational time

of the CGS algorithm and the cWY algorithm are shorter and $\|AV - VD\|_F$ and $\|V^\top V - I\|_F$ of the cWY algorithm are sufficiently small.

In a future work, the inverse iteration algorithms should be compared on General-purpose computing on graphics processing units (GPGPU).

Table 2: Experimental results

	algorithm	time[sec.]	$\|AV - VD\|_F$	$\|V^\top V - I\|_F$
type1	(dimension size is 2100.)			
	CGS	10.35	9.15×10^{-15}	2.50×10^{-14}
	MGS	7.32	9.15×10^{-15}	2.50×10^{-14}
	cWY	13.51	0.70×10^{-15}	2.61×10^{-14}
	(dimension size is 4200.)			
	CGS	60.54	1.25×10^{-14}	3.31×10^{-14}
	MGS	64.51	1.25×10^{-14}	3.32×10^{-14}
	cWY	94.27	0.067×10^{-14}	3.36×10^{-14}
	(dimension size is 6300.)			
	CGS	188.52	1.52×10^{-14}	3.49×10^{-14}
	MGS	478.53	1.53×10^{-14}	3.49×10^{-14}
	cWY	318.04	0.30×10^{-14}	4.52×10^{-14}
	(dimension size is 8400.)			
	CGS	768.40	1.82×10^{-14}	3.47×10^{-14}
	MGS	5887.12	1.81×10^{-14}	3.47×10^{-14}
	cWY	1408.27	1.01×10^{-14}	21.48×10^{-14}
type2	(dimension size is 2100.)			
	CGS	43.15	8.72×10^{-14}	1.06×10^{-13}
	MGS	263.82	8.64×10^{-14}	1.11×10^{-13}
	cWY	78.75	0.37×10^{-14}	2.56×10^{-13}
	(dimension size is 4200.)			
	CGS	247.92	1.79×10^{-13}	1.84×10^{-13}
	MGS	2392.14	1.77×10^{-13}	1.97×10^{-13}
	cWY	456.93	0.052×10^{-13}	4.96×10^{-13}
	(dimension size is 6300.)			
	CGS	754.69	2.64×10^{-13}	2.83×10^{-13}
	MGS	7864.63	2.63×10^{-13}	3.04×10^{-13}
	cWY	1394.79	0.061×10^{-13}	7.45×10^{-13}
	(dimension size is 8400.)			
	CGS	1718.53	3.51×10^{-13}	4.33×10^{-13}
	MGS	16770.71	3.48×10^{-13}	4.53×10^{-13}
	cWY	3186.58	0.078×10^{-13}	10.18×10^{-13}
type3	(dimension size is 2100.)			
	CGS	20.13	1.11×10^{-12}	1.00×10^{-13}
	MGS	28.15	1.11×10^{-12}	1.07×10^{-13}
	cWY	35.64	0.18×10^{-13}	1.06×10^{-13}
	(dimension size is 4200.)			
	CGS	89.47	1.75×10^{-12}	7.72×10^{-12}
	MGS	202.16	1.75×10^{-12}	1.53×10^{-12}
	cWY	158.18	0.25×10^{-12}	0.95×10^{-12}
	(dimension size is 6300.)			
	CGS	210.98	2.37×10^{-11}	77.29×10^{-10}
	MGS	678.24	2.51×10^{-11}	2.00×10^{-10}
	cWY	371.78	26.89×10^{-11}	2.17×10^{-10}
	(dimension size is 8400.)			
	CGS	391.50	7.93×10^{-12}	757.15×10^{-11}
	MGS	1422.99	7.94×10^{-12}	3.13×10^{-11}
	cWY	678.31	3.13×10^{-12}	3.13×10^{-11}

References

[1] (2012) Basic Linear Algebra Subprograms. [Online]. Available: http://http://netlib.org/blas/index.html

[2] (2012) Cell SDK Installation Guide. [Online]. Available: http://git.gitbrew.org/openclit/documentation/SDK-Installation_Guide_v3.1.pdf

[3] J. J. M. Cuppen, "A divide and conquer method for the symmetric tridiagonal eigenproblem," *N*umerische Mathematik, Vol. 36, pp. 177–195, 1981.

[4] J. Demmel, *A*pplied Numerical Linear Algebra, Philadelphia America: SIAM, 1997.

[5] J. Demmel, L. Grigori, M. Hoemmen and J. Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, LAPACK Working Notes, No.204, 2008.

[6] J. Demmel, O. Marques, B. Parlett, and C. Vömel, "Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers," *S*IAM J. Sci. Comput., Vol. 30, No. 3, pp. 1508–1526, 2008.

[7] I. Dhillon, *A new O(n^2) algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem*, Ph.D. thesis, Computer Science Division, University of California, Berkeley, California, available as UC Berkeley Technical Report UCB//CSD-97-971, 1997.

[8] I. Dhillon, B. Parlett, and C. Vömel, "Glued matrices and the MRRR algorithm," *S*IAM J. Sci. Comput., Vol. 27, No. 2, pp. 496–510, 2005.

[9] G. Golub and W. Kahan, "Calculating the singular values and pseudo-inverse of a matrix," *S*IAM J. Numeri. Anal., Vol. 2, pp. 205–224, 1965.

[10] G. Golub and C. van Loan, *Matrix Computations*, Maryland, America: Johns Hopkins University Press, 1996.

[11] A. Greenbaum, M. Rozloznik, and Z. Strakos, "Numerical Behaviour of The Modified Gram-Schmidt GMRES Implementation," *B*IT, No. 69, pp.303–318, 1996.

[12] M. Gu and S. C. Eisenstat, *A stable algorithm for the rank-1 modification of the symmetric eigenproblem*, Computer Science Department Report YALEU/DCS/RR-916, Yale University, New Haven, CT, 1992.

[13] M. Gu and S. C. Eisenstat, "A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem," *S*IAM Journal on Matrix Analysis and Applications, Vol. 16, No. 1, pp.172–191, 1995.

[14] (2012) IBM United States. [Online]. Available: http://www.ibm.com/

[15] H. Ishigami, K. Kimura, and Y. Nakamura, "Implementation and performance evaluation of new inverse iteration algorithm with Householder transformation in terms of the compact WY representation" in *P*roc. PDPTA2011, 2011, Vol. II, pp. 775-781.

[16] (2012) LAPACK-Linear Algebra PACKage. [Online]. Available: http://www.netlib.org/lapack/

[17] G. Peters and J. Wilkinson, *The calculation of specified eigenvectors by inverse iteration*, contribution II/18, in Linear Algebra, Handbook for Automatic Computation, Vol. II, Springer-Verlag, Berlin, pp. 418-439, 1971.

[18] R. Schreiber and C. van Loan, *A storage-efficient WY representation for products of Householder transformations*, SIAM J. Sci. Stat. Comput., Vol. 10, No. 1, pp. 53-57, 1988.

[19] H. Walker, *Implementation of the GMRES method using Householder transformations*, SIAM J. Sci. Stat. Comput., Vol. 9, No. 1, pp. 152–163, 1988.

[20] Y. Yamamoto and Y. Hirota, *A parallel algorithm for incremental orthogonalization based on the compact WY representation*, JSIAM Letters, Vol. 3, pp. 89–92, 2011.

Automatic Generation of Diagram Explanation based on an Attribute Graph Grammar

Takaaki Goto[1], **Tetsuro Nishino**[1], **and Kensei Tsuchida**[2]
[1]Graduate School of Informatics and Engineering, The University of Electro-Communications, Chofu, Tokyo, Japan
[2]Faculty of Information Science and Arts, Toyo University, Kawagoe, Saitama, Japan

Abstract— *Unified Modeling Language (UML) has already been used in the analysis, design, and implementation of many systems. Open Source Software (OSS) is often used in software development. However, it is often the case that OSS does not contain adequate documents, so the generation of software documents is important. Documents with diagrams are especially important to understand software, so it is important to generate documents with diagrams. In order to process large-scale diagrams, or many source codes automatically, formal and declarative representation is needed. In this paper, we propose automatic generation of documentation based on an attribute graph grammar.*

Keywords: UML, graph grammar, open source software, software document

1. Introduction

In the software development environment, it is desirable to have features that support programming. Many effective tools have been developed to provide a framework for developing reliable programs. In software development, software documents are very important to develop or modify systems. Graphical representations such as flowchart or Unified Modeling Language (UML) are often used in software design and development because of their expressiveness.

UML for modeling in software development has been proposed recently. In 2005, ISO/TEC 19501 became the standard. UML has already been used in the analysis, design, and implementation of many systems. It makes use of various types of diagrams, such as class and sequence diagrams, for designing processes in system development, from upstream to downstream processes. In order to automate the processing of these graphical representations using computers, a syntax for program diagrams must first be defined. Then, in order to analyze the syntax of two-dimensional objects such as program diagrams, the relationships between each of the elements must also be described. Graph grammars are one possible effective means for implementing these methods. Graph grammars provide a formal method that enables rigorous definition of mechanisms for generating and analyzing graphs. The authors of the current paper have proposed a graph grammar for package diagrams of UML [1].

Open Source Software (OSS) is often used in software development. However, it is often the case that OSS does not contain adequate documents. Software documents are needed in order to use or modify OSS, however, there are so many examples of source code with no documents, therefore, we need to generate documents from source code. Moreover, documents with diagrams are especially important to understand software, so it is important to generate documents with diagrams. This is why we started our research so that we can obtain documents with diagrams automatically.

Thus far, much research has targeted UML or software documents. Research has also been done on UML [2] and graph grammars and graph transformations with respect to UML [3], [4], [5]. However, no research focuses on syntax formalization for visual representation. As for software document generation, some research has been done [6], [7], [8]. This research targets documentation generation through program source code or diagrams. The formal method is not treated in this framework.

In order to process large-scale diagrams, or large amounts of source code automatically, formal and declarative representation is needed. In this paper, we propose automatic generation of documentation based on an attribute graph grammar.

2. Preliminary

2.1 Graph Grammars

Definition 1. ([9],[10]) Let Σ be an alphabet of node labels and Γ be an alphabet of edge labels. A *graph* over Σ and Γ is a tuple $H = (V, E, \lambda)$, where V is the finite set of nodes, $E \subseteq \{(v, \gamma, w) \mid v, w \in V, v \neq w, \gamma \in \Gamma\}$ is the set of edges, and $\lambda : V \to \Sigma$ is the node labeling function. $E(v, w) \stackrel{\text{def}}{=} \{\gamma \in \Gamma \mid (v, \gamma, w) \in E\}$. The *label tuple* of two nodes $v, w \in V$ is $lab(v, w) \stackrel{\text{def}}{=} (\lambda(v), E(v, w), E(w, v), \lambda(w))$. □

Definition 2. ([9]) A *graph with* (neighborhood controlled) *embedding* over Σ and Γ is a pair (H, C) with $H \in GR_{\Sigma,\Gamma}$ and $C \subseteq \Sigma \times \Gamma \times \Gamma \times V_H \times$ {in, out}. C is the *connection relation* of (H, C), and each element $(\sigma, \beta, \gamma, x, d)$ of C (with $\sigma \in \Sigma, \beta, \gamma \in \Gamma, x \in V_H$, and $d \in$ {in, out}) is a *connection instruction* of (H, C). A connection instruction

$(\sigma, \beta, \gamma, x, d)$ will always be written as $(\sigma, \beta/\gamma, x, d)$. Two graphs with embedding (H, C_H) and (K, C_K) are isomorphic if there is an isomorphism f from H to K such that $C_K = \{(\sigma, \beta/\gamma, f(x), d) \mid (\sigma, \beta/\gamma, x, d) \in C_H\}$. The set of all graphs with embedding over Σ and Γ is denoted as $GRE_{\Sigma,\Gamma}$. □

Figure 1 shows an example of a graph. In Figure 1 $H = (V_H, E_H, \lambda_H)$ is a graph with $V_H = \{n_1, n_2\}$, $E_H = \{(n_1, \alpha, n_2)\}$, $\lambda_H(n_1) = a$ and $\lambda_H(n_2) = X$. Here, n_1 and n_2 indicate node ID. Furthermore a, and X indicate node labels, nodes with a lowercase node label and with a uppercase node label are terminal node and nonterminal node, respectively,

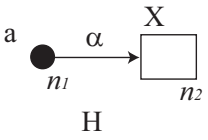


Fig. 1: An example of a graph

Definition 3. ([9]) An *edNCE graph grammar* is a six-tuple $GG = (\Sigma, \Delta, \Gamma, \Omega, P, S)$, where Σ is the alphabet of node labels, $\Delta \subseteq \Sigma$ is the alphabet of terminal node labels, Γ is the alphabet of edge labels, $\Omega \subseteq \Gamma$ is the alphabet of final edge labels, P is the finite set of *productions*, and $S \in \Sigma - \Delta$ is the *initial nonterminal*. A production is of the form $X \to (D, C)$ where X is a nonterminal node label, D is a graph over Σ and Γ, and $C \subseteq \Sigma \times \Gamma \times \Gamma \times V_D \times \{in, out\}$ is the connection relation, which is a set of connection instructions. A pair (D, C) is a graph with embedding over Σ and Γ. □

Definition 4. (cf. [9], [10]) Let $G = (\Sigma, \Delta, \Gamma, \Omega, P, S)$ be an edNCE graph grammar. Let $H_{i-1} = (V_{H_{i-1}}, E_{H_{i-1}}, \lambda_{H_{i-1}})$ and $H_i = (V_{H_i}, E_{H_i}, \lambda_{H_i})$ be graphs in $GRE_{\Sigma,\Gamma}$. In addition, let $v_i \in V_{H_{i-1}}$, and $p'_i : X \to (D'_i, C'_i) \in P$ be a production copy of G such that D'_i and H_{i-1} are disjoint. $s_i = (p'_i, v_i, D'_i, b'_i)$ is a *derivation specification* of G if $p'_i \in copy(P)$, $\lambda_{H_{i-1}(v_i)} = X$, $D'_i \cong D$, $b'_i : V_{D'_i} \to V_{D_i}$.

We write $H_{i-1} \to_{v_i, p'_i} H_i$, or just $H_{i-1} \underset{s_i}{\to} H_i$, if $\lambda_{H_{i-1}}(v_i) = X$ and $H_i = H_{i-1}[v_i/(D'_i, C'_i)]$. $H_{i-1} \underset{s_i}{\to} H_i$ is called a *derivation step*, and a sequence of such derivation steps is called a *derivation*. □

An example of a production is shown in Figure 2. In the figure, a box is a nonterminal node and a filled circle is a terminal node. X, Y, and b are node labels and v_0, v_1, and v_2 are node IDs. Nodes with the same node label can appear in a graph, while nodes with same node ID will never appear in a graph. The production of Figure 2 indicates that after the removal of a nonterminal node with label X, embed the graph consisting of terminal node with label b and the nonterminal node with label Y. Each production has connection instructions. The connection instruction of this production is $(a, \alpha/\beta, v_1, in)$, but this connection instruction is not described in the notation of Figure 2.

In Figure 3, the production of Figure 2 and its connection instruction are drawn simultaneously. The large box in Figure 3 indicates the left-hand side, and two nodes with label b and Y are on the right side of the production of Figure 2. Node labels and edge labels that are described outside of the large box indicate connection instruction such that $(a, \alpha/\beta, v_1, in)$.

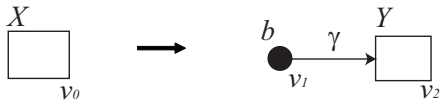


Fig. 2: An example of a production

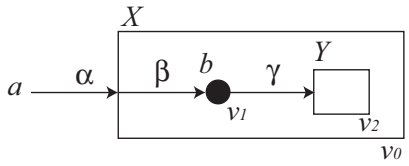


Fig. 3: An example of a production with the connection relation

An example of application of the production is shown in Figure 4. In Figure 4, $H = (V_H, E_H, \lambda_H)$ is a graph with $V_H = \{n_1, n_2\}$, $E_H = \{(n_1, \alpha, n_2)\}$, $\lambda_H(n_1) = a$, and $\lambda_H(n_2) = X$. The production copy p' of p is as follows: $p' : X \to (D', C')$ where $X = \lambda_H(n_2)$, $D' = (V_{D'}, E_{D'}, \lambda_{D'})$ such that $V_{D'} = \{n_3, n_4\}$, $E_{D'} = \{(n_3, \gamma, n_4)\}$, $\lambda_{D'}(n_3) = b$, $\lambda_{D'}(n_4) = Y$, and $C' = \{(a, \alpha/\beta, n_3, in)\}$.

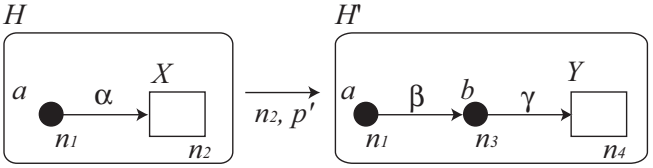


Fig. 4: An example of applying a production rule

We say that H is the *host graph* and H' is the *resulting graph*, X is the *mother node* in Figure 4, the graph consisting of terminal node with label b and the nonterminal node with label Y in Figure 2 is the *daughter graph*. At first, we remove the node X and edges that connect with node X from host graph H. Next, we embed the daughter graph, including node b and node Y. Then we establish edges between the nodes of daughter graph and the nodes that were connected to the node X using the connection instructions on the production p'. Therefore, the edge label α is rewritten to β by the production p'.

Definition 5. ([11], [12]) An *Attribute edNCE Graph Grammar* is a three-tuple $AGG = \langle GG, Att, F \rangle$, where

1. $GG = (\Sigma, \Delta, \Gamma, \Omega, P, S)$ is called an *underlying graph grammar* of AGG. Each production p in P is denoted by $X \rightarrow (D, C)$.

2. Each node symbol $Y \in \Sigma$ of GG has two disjoint finite sets $Inh(Y)$ and $Syn(Y)$ of *inherited* and *synthesized attributes*, respectively. The set of all attributes of symbol X is defined as $Att(X) = Inh(X) \cup Syn(X)$. $Att = \bigcup_{X \in \Sigma} Att(X)$ is called the *set of attributes* of AGG. We assume that $Inh(S) = \emptyset$. An attribute a of X is denoted by $a(X)$, and the set of possible values of a is denoted by $V(a)$.

3. Associated with each production $p = X_0 \rightarrow (D, C) \in P$ is a set F_p of *semantic rules*, which define all the attributes in $Syn(X_0) \bigcup_{X \in Lab(D)} Inh(X)$. A semantic rule defining an attribute $a_0(X_{i_0})$ has the form $a_0(X_{i_0}) := f(a_1(X_{i_1}), \cdots, a_m(X_{i_m}))$. Here f is a mapping from $V(a_1(X_{i_1})) \times \cdots \times V(a_m(X_{i_m}))$ into $V(a_0(X_{i_0}))$. In this situation, we say that $a_0(X_{i_0})$ depends on $a_j(X_{i_j})$ for j, $0 \leq j \leq m$ in p. The set $F = \bigcup_{p \in P} F_p$ is called the *set of semantic rules* of G. □

Attribute values are calculated by evaluating attributes according to semantic rules on the derivation tree.

2.2 UML

Unified Modeling Language (UML) is a notation for modeling object-oriented system development using diagrams. UML can be divided into structural diagrams and behavioral diagrams. Structural diagrams are used to describe the structure of what is being modeled and include class, object, and package diagrams. Behavioral diagrams are used to describe the behavior of what is being modeled and include use-case, activity, and state-machine diagrams.

Structural diagrams include class diagrams, which describe the static relationships between classes, and package diagrams, which group classes and describe relationships between packages and package nesting relationships.

Figure 5 shows an example of a package diagram. The box with a rectangle at the upper left indicates a package. The box with three compartments is a class. Each of the three parts indicates its class name, its attribute, and its methods from top to the bottom. A plus with a circle is used to represent which components the package contains. Package 1 contains Package 2 and Package 4, and Package 4 contains Class 1 and Class 2.

3. Graph Grammar for UML Package Diagrams

In this section, we describe our Graph Grammar for Package Diagrams (GGPD), for UML package diagrams.

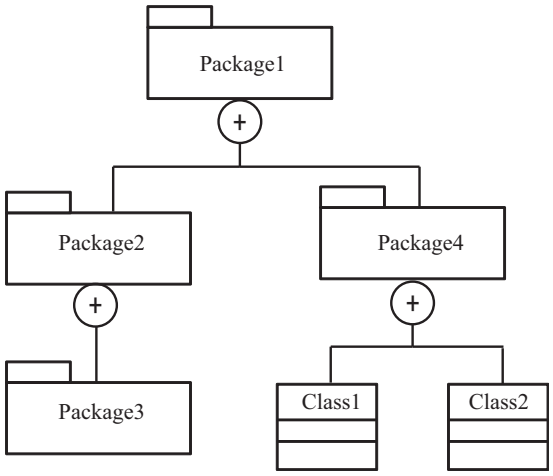


Fig. 5: An example of a package diagram

3.1 Grammar Overview

Definition 6. The Graph Grammar for Package Diagrams (GGPD), for UML package diagrams, is a six-tuple $GGPD$ = (Σ_{PD}, Δ_{PD}, Γ_{PD}, Ω_{PD}, P_{PD}, S_{PD}). Here, Σ_{PD} = { S, A, T, L, R, M, rop, sp, lep, rip, mip, lec, mic, ric } is a finite set of node labels, Δ_{PD} = { rop, sp, lep, rip, mip, lec, mic, ric } is a finite set of terminal node labels, Γ_{PD} = { $*$ }, Ω_{PD} = { $*$ }, P_{PD} = { $P_1, ..., P_{17}$ } is a finite set of production rules, and S_{PD} = { S } is the initial non-terminal. □

The GGPD generates package hierarchy diagrams. Terminal nodes generated by GGPD have the following node labels: rop (root of package), sp (single package), lep (left side package), rip (right side package), mip (package located between lep and rip), lec (left side class), ric (right side class), and mic (class located between lec and ric).

GGPD is a context-free grammar and there are 17 production rules. An example of GGPD production rule is shown in Figure 6.

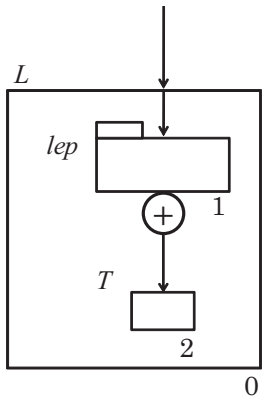


Fig. 6: An Example of a production rule of GGPD

In the figure, the production rule can be applied to a node labeled L, which is a non-terminal node, to generate a terminal node with the label lep, representing a package, and a non-terminal node labeled T.

A node with a capitalized label indicates a nonterminal node, and a node with an uncapitalized label indicates a terminal node. Our grammar generates directed graphs. However, we drew the graphs without arrows as we assumed the direction of each edge was from the top down.

We omit descriptions of all of the production rules because of space limitations.

3.2 Example of Derivation

Figure 7 shows an example of a GGPD derivation. In this example, G_0 is a graph with the node labeled S. The node ID is 1 (lower right of the node).

Then the production rule P_1 is applied to a non-terminal node labeled S with node ID 1, which is the initial non-terminal node. That is, remove a mother node with label S and node ID 1, then embed a daughter graph in the P_1. In this case, the daughter graph is the node with label A. This produces the non-terminal node labeled A with node ID 2, to which the P_3 production rule is applied. That is, graph G_1, which consists of node with node ID 2, is obtained.

After application of the production P_3, the terminal node labeled rop and a non-terminal node labeled T are generated. We apply productions to obtain a graph that corresponds to UML package diagrams. In this case, we can obtain graph G_9.

We can obtain a derivation tree from a derivation sequence of production. Figure 8 shows the derivation tree corresponding to Figure 7. In Figure 8, the labels show the names of the production rules.

4. Document Generation

In this section, we explain some attributes and semantic rules for document generation. In this paper, we target a documentation that has diagrams and explanations that correspond to diagrams. We generate documents by attribute evaluation that can be executed automatically on derivation trees.

We can obtain derivation trees after generating diagrams based on our grammar. Figure 8 shows an example of a derivation tree. In the Figure, for example, Production 3 (P3) is applied to a node that was generated by Production 1 (P1). Derivation trees describe a process of applying productions.

Every node generated by productions of GGPD has some attributes. An attribute has two types of values called inherited attributes and synthesized attributes. Attribute values are obtained by calculating semantic rules on derivation trees. Values of inherited attributes are calculated by top-down on derivation trees, and bottom-up calculation generates values of synthesized attributes.

Figure 9 shows an example of production rules and their corresponding semantic rules. The upper part of Figure 9 indicates a production rule and the lower part shows semantic rules. This production rule rewrites a nonterminal node with node label R to a graph consisting of terminal node rip and nonterminal node T.

In the semantic rules part, we then find three semantic rules that process the values of local and global attributes. The global attribute preserves the entire explanation of the diagrams. The local attribute retains local information such as parent and child relations. These attributes are categorized as synthesized attributes.

In Figure 9, local(1) stores node name of the node with ID 1 (in this case, package diagram's name); the value of local(1) is assigned to local(0).

Figure 10 shows an example of a derivation tree with document generation semantic rules corresponding to Figure 7.

In order to obtain documentation, attribute evaluation is processed following a bottom up order $P_4 \rightarrow P_9 \rightarrow P_{10} \rightarrow P_{14} \rightarrow P_6 \rightarrow P_{13} \rightarrow P_6 \rightarrow P_3 \rightarrow P_1$.

First, P_4 located at the lower left on the derivation tree is processed. In this case, we assume that the package name is "package 9," which is substituted for the local attribute of the node with ID 9. Local(9) is substituted for local(8). Null is substituted for global(9)

Next, P_9 is processed. Here, package name is substituted for local(7), and local(5) has the value of local(7). The global(5) stores a partial explanation of diagrams. In this case, global(5) holds information that node with ID 7 in Figure 7 has packages. The global(5) has the following sentence: "package 7 has package 9".

Global value is similarly obtained from the above procedure. The global(1) explains the entire diagram; in this example, global(1) is explained in Figure 11.

Since semantic rules are declarative defined, we can obtain an explanation corresponding to diagrams if once we define the semantic rules.

5. Conclusion

In this paper, we have defined attributes and semantic rules for a graph grammar for UML package diagrams. We can generate documents corresponding to diagrams automatically according to a declarative definition.

A future issue for study is the synchronization between diagram and documentation and the implementation of systems that can generate documents with animation. We would also like to construct a graph grammar for other diagrams in UML.

References

[1] Takaaki Goto, Tetsuro Nishino, Kensei Tsuchida, "An Attribute Graph Grammar for UML Package Diagrams and its Applications," in *Proceedings of The 2011 International Conference on Parallel and*

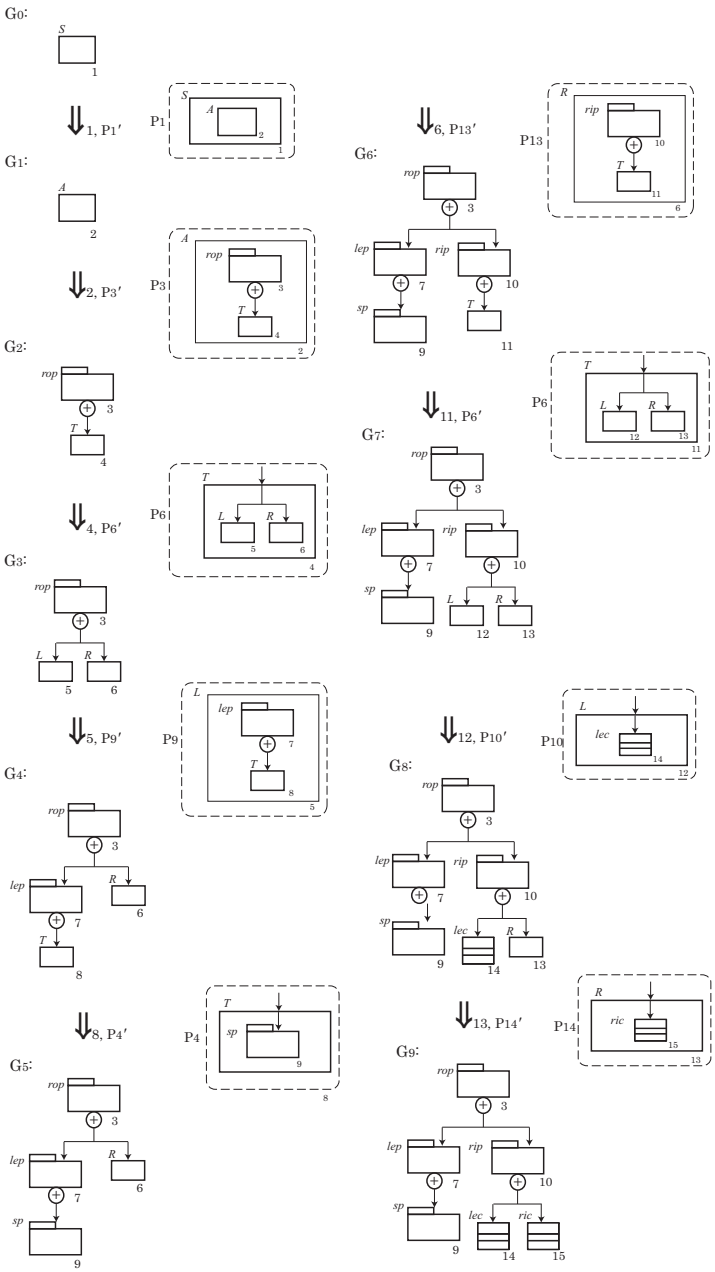


Fig. 7: An example of a GGPD derivation

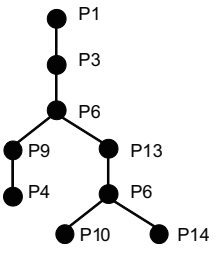


Fig. 8: A derivation tree corresponding to the tree in Figure 7

Distributed Processing Techniques and Applications, Volume II, 2011, pp. 693–698.

[2] L. Kotulski and D. Dymek, "On the Modeling Timing Behavior of the System with UML(VR)," in *Computational Science ICCS 2008*, ser. Lecture Notes in Computer Science, vol. 5101, 2008, pp. 386–395.

[3] F. Hermann, H. Ehrig, and G. Taentzer, "A Typed Attributed Graph Grammar with Inheritance for the Abstract Syntax of UML Class and Sequence Diagrams," *Electron. Notes Theor. Comput. Sci.*, vol. 211, pp. 261–269, April 2008.

[4] Kong, Jun and Zhang, Kang and Dong, Jing and Xu, Dianxiang, "Specifying behavioral semantics of UML diagrams through graph transformations," *J. Syst. Softw.*, vol. 82, pp. 292–306, 2009.

[5] D. Petriu and H. Shen, "Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Spec-

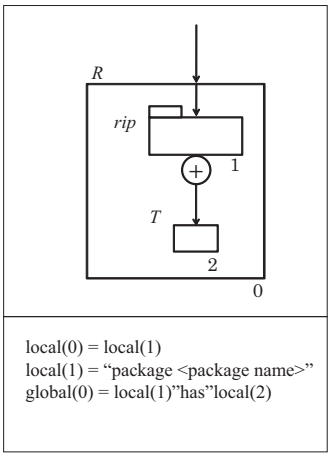


Fig. 9: An Example of an attribute for document generation of GGPD

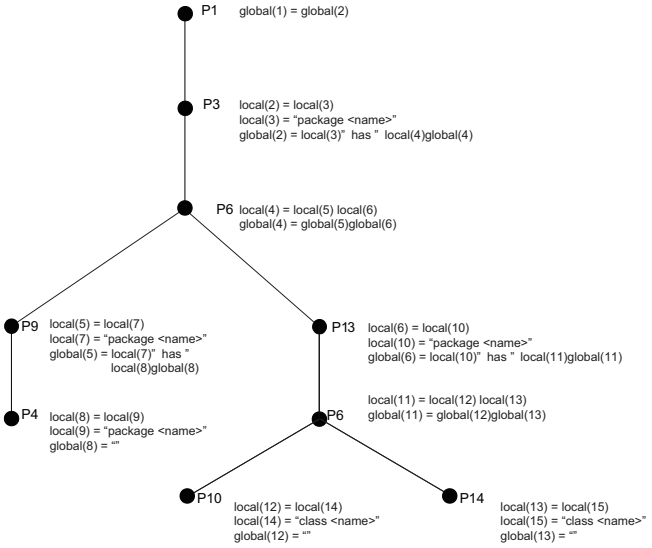


Fig. 10: A derivation tree with document generation attributes

Package 2 has package 3.
Package 4 has class 5, class6.
Package 1 has package 2, package 4.

Fig. 11: An example of obtained explanation

ifications,” in *Computer Performance Evaluation: Modelling Techniques and Tools*, ser. Lecture Notes in Computer Science, vol. 2324, 2002, pp. 183–204.

[6] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, “Towards automatically generating summary comments for java methods,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE ’10. New York, NY, USA: ACM, 2010, pp. 43–52.

[7] F. Meziane, N. Athanasakis, and S. Ananiadou, “Generating natural language specifications from uml class diagrams,” *Requirements Engineering*, vol. 13, pp. 1–18, 2008, 10.1007/s00766-007-0054-0.

[8] J. W. Nimmer and M. D. Ernst, “Automatic generation of program specifications,” *SIGSOFT Softw. Eng. Notes*, vol. 27, pp. 229–239, July 2002.

[9] G. Rozenberg, *Handbook of Graph Grammar and Computing by Graph Transformation Volume 1*. World Scientific Publishing, 1997.

[10] M. Kaul, “Practical applications of precedence graph grammars,” in *Graph Grammars and Their Application to Computer Science*, ser. LNCS 291, 1986, pp. 326–342.

[11] T. Nishino, “Attribute Graph Grammars with Applications to Hichart Program Chart Editors,” in *Advances in Software Science and Technology*, vol. 1, 1989, pp. 89–104.

[12] T. Arita, K. Sugita, K. Tsuchida, and T. Yaku, “Syntactic Tabular Form Processing by Precedence Attribute Graph Grammars,” in *Proc. IASTED Applied Informatics 2001*, 2001, pp. 637–642.

Modeling the Component Pickup and Placement Sequencing Problem with Nozzle Assignment in a Chip Mounting Machine

Hiroaki Konishi, Hidenori Ohta and Mario Nakamori
Department of Information and Computer Sciences, Faculty of Engineering
Tokyo University of Agriculture and Technology
Koganei, Tokyo, Japan

Abstract - *In the production of electronic circuit boards in industry, electronic components are placed onto the circuit board by component placement machines. The actions of multifunction placement machines are roughly divided into pick-up phase and placing phase. In the past, these two phases were typically optimized independently of each other to be combined, which do not necessarily lead to the overall optimization. Therefore, in this paper, the action of multifunction component placement machines is modeled in a different way than that in the conventional approach, and a method for comprehensive optimization of pick-up phase and placing phase is proposed. The proposed method, which takes into account nozzle allocation, can also be applied to the load balancing of component placement machine production lines.*

Keywords: component placement machine, line-balancing

1 Introduction

Electronic circuit boards are produced through several processes (printing, placing, heat treatment, etc.), among which the process of placing the components onto the circuit boards is the most significant and often becomes the bottleneck. In this process, components are automatically placed onto the circuit boards by machines called component placement machines. Since the circuit board production efficiency is dependent on the performance of the component placement machines, a variety of research has been published on motion optimization and operation of each of the several types of component placement machines already developed. However, a relatively new type of component placement machines called *multifunction component placement machines* have a large degree of freedom of motion, and only a few researches are reported on optimization algorithms for this type of machines. In the present paper we consider multifunction component placement machines and call hereafter "component placement machines" in abbreviation.

1.1 Component placement machines

At circuit board production sites in industry, production is usually carried out through production lines composed of multiple component placement machines connected in series. More specifically, it is determined in advance which components will be assigned to each component placement machine (machine allocation). When the placing of components allocated to one component placement machine is complete, the circuit board is carried to the next component placement machine by a conveyor. To place all of the components on a circuit board, it is necessary for the circuit board to pass through all component placement machines in the line.

A simplified schematic of a component placement machine is shown in Figure 1. The component placement machine has a *head* which has multiple *nozzles* for picking up and placing components.

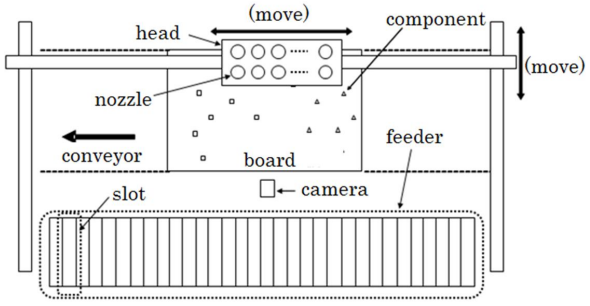


Figure1 Component placement machine

The component placement machine places components on a circuit board by repeating the following actions:

(1) The head moves to a position above the component feeder and the nozzles pick up the required components from the appropriate feed slots;
(2) The head moves to the camera which checks whether the components have been picked up correctly;
(3) When it has been confirmed that the components have been picked up correctly, the head moves above the circuit board and travels around the board, placing the components in their respective places;
(4) When all of the picked-up components have been placed, the head returns to the initial position above the component feeder.

The abovementioned sequence of actions is called a *turn*. Generally, multiple turns are carried out before all components allocated to one component placement machine have been placed.

The time required for placing components in a turn is approximately proportional to the travel distance of the head. The head travel distance is shorter if picking-up and placing positions are closer together. In order to optimize the picking up and placing actions, it is important to assign components to turns so that the total sum of travel distance of the head is the minimum.

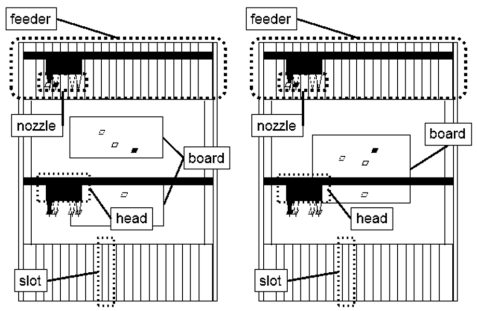
1.2 Mounting method of component placement machines

There are two typical methods of mounting for component placement machines; *alternating mounting* and *separate mounting*.

In the alternating mounting method, two heads positioned opposite each other alternately place components onto a central circuit board, and while one head is performing a component placing action, the other head performs a component picking-up action. Since placing usually requires more time than picking up, the motion time of the component placement machine is determined only by the component placing action.

In the separate mounting method, on the contrary, only one head places the components onto one circuit board; there are two conveyors, which carry two boards independently on each other. Unlike the alternating mounting method, the component placement machine's motion time in the separate mounting method is determined by both the component placing action and the pick-up action.

Most component placement machines are equipped with two conveyors, and it is possible to carry out separate mounting using two conveyors as well as alternating mounting using only one conveyor; recently, however, the most common placing method used at circuit board production sites has been shifting from alternating mounting to separate mounting. Figure 2 shows a simplified schematic of each mounting method.



(a) (b)

Figure 2 Separate mounting (a) and alternate mounting (b)

The component placement machine shown in Figure 1 is just one side of the component placement machine shown in Figure 2 (i.e., lower half of Figure 2). In the following, "one machine" refers to one side of a machine that uses the separate mounting method (upper or lower half).

1.3 Problems concerning motion optimization of component placement machines

In both the pick-up action and the placing action, the travel distance of the head may vary greatly depending on which components are allocated to which nozzle (*nozzle allocation*), because the size of the head is not negligible compared to the size of the circuit board and the component feeder. Figure 3 shows how the travel distance of the head when placing components varies by nozzle allocation.

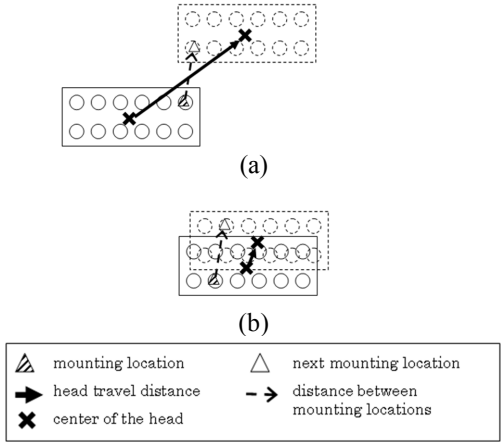


Figure3 Difference of travel distance by nozzle allocation

Also, to increase the speed of the pick-up action, recent component placement machines are capable of picking up multiple components at the same time (*gang pick*), if the interval of nozzle positions and that of slot positions of the concerning components are the same. For example, with the arrangement of the component feeder shown in Figure 4, it is possible for the two components supplied from slots 5 and 7 to be picked up simultaneously by nozzles B and D, respectively. Gang pick reduces the number of component picking-ups and accelerates the picking-up action.

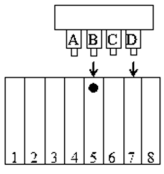


Figure4 Gang pick

Thus, considering the problem of nozzle allocation is very important.

In the former research, however, nozzle allocation was not often given sufficient consideration. Yamamoto et al. [3] proposed a method for optimizing the placing action taking the nozzle allocation into account and achieved a reduction in head travel distance of approximately 20% compared to conventional methods. This research, however,

considered the placing action only and neglected the pick-up action. When the placing time is longer than the pick-up time in the alternating mounting method, this approach is adequate. On the contrary, in the separate mounting method, both placing time and pick-up time are reflected in the circuit board production time, and so the assumption of the above approach does not hold. At actual circuit board production sites in industry, when motion optimization of component placement machines under the separate mounting method is performed, nozzle allocation is determined from the point of view of optimization of the picking-up action first and optimization of the placing action is carried out next. The method proposed by Yamamoto et al. can cope with the alternating mounting method, but it cannot sufficiently optimize component placement machines under the separate mounting method.

In this paper, we propose first a model for simultaneous optimization of picking-up action and placing action by considering nozzle allocation in one component placement machine under the separate mounting method.

At actual sites where component placement machines are in operation, multiple machines are connected to form a production line. Machine allocation, turn allocation, nozzle allocation, component pick-up action, and component placing action should really be optimized in a comprehensive way that also takes into account line balancing. As for line balancing, a method for allocating components by finding placing paths in component placement machines has been proposed, but this method does not consider nozzle allocation or component pick-up action [2].

The model proposed in this paper is basically intended for motion optimization of one machine, but line balancing can be carried out at the same time by using a simple extension.

2 Formulation of the problem

In this paper, we consider the following two problems.

2.1 Problem of motion optimization of one machine

This is the problem of finding turn allocation, nozzle allocation, pick-up order, and placing order for given components, with the aim of carrying out motion optimization of one component placement machine on a production line, where components are already allocated to machines. In this paper, the arrangement of the slots in the component feeder is given, and it is assumed that one type of component is supplied from exactly one feed slot and the same type of component is not placed in multiple slots. Also, in real component placement machines, the type of component that can be picked up and placed may be limited according to the type of nozzle, but, here, it is assumed that all types of components can be picked up and placed by any nozzle.

The motion time spent on each circuit board by the component placement machine is approximately proportional to the travel distance of the head, and so the head travel distance is used to evaluate the solution. However, to take into account the number of component pick-ups, the time spent for picking up is converted to head travel distance and added to the evaluation value. Furthermore, because the head is driven by motors operating independently in the x- and y-directions, the head travel distance is defined as the Chebyshev distance.

2.2 Problem of motion optimization of multiple machines and line balancing

This is the problem of load balancing between machines on a production line composed of multiple machines. This is an extension of the problem of motion optimization of a single machine. In this problem, it is necessary to find the machine allocation of components as well as turn allocation, nozzle allocation, pick-up order, and placing order for each machine.

The amount of time spent per circuit board in the production line is the motion time of the machine that causes bottlenecking, so the solution is evaluated by using the head travel distance of the bottleneck machine. In the same way as the problem of motion optimization of one machine, the time spent picking up is converted to distance and added to the evaluation value.

3 The algorithm

In our proposed algorithm, component turn allocation is sought in combination with nozzle allocation using a local search technique. Also, these allocations are evaluated by finding the pick-up and placing paths for each allocation and calculating the respective head travel distance and the number of pick-ups.

A flowchart of the proposed algorithm is shown in Figure 5.

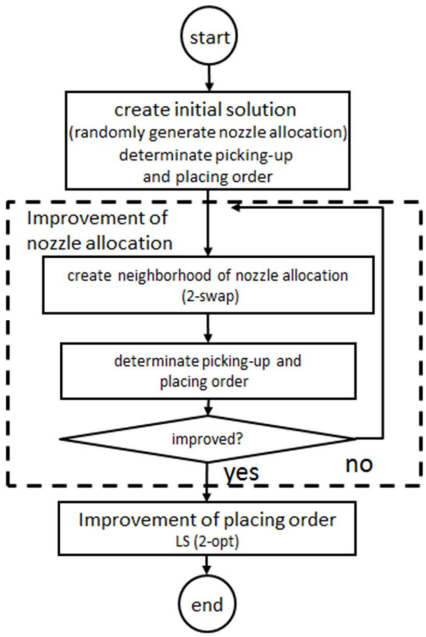


Figure5 Flowchart of the proposed algorithm

3.1 Method of expression of nozzle allocation

3.1.1 Expression of nozzle allocation for a single machine

Individual numbers are assigned to the components, and nozzle allocation is expressed together with component turn allocation by arranging the assigned component numbers, as shown in Figure 6. Component turn allocation is obtained by dividing up the arranged components based on the number of nozzles in the head, starting from the leftmost component. Also, the sequence of component numbers within a turn indicates the nozzle picking them up. Figure 6 shows an example of nozzle allocation in a component placement machine with four nozzles. The character "e" is inserted instead of a component number when the nozzle does not pick up a component.

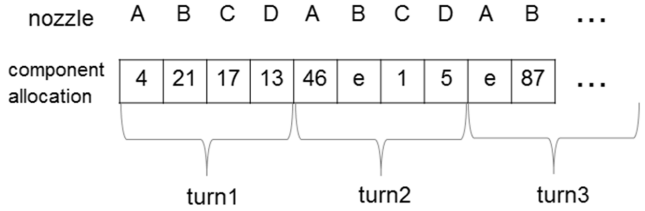


Figure6 Nozzle allocation in a placement machine

3.1.2 Expression of nozzle allocation for multiple machines and line balancing

As in the expression of turn allocation and nozzle allocation in the problem of motion optimization of a single machine, individual numbers are assigned to the components, where nozzle allocation is expressed together with component machine allocation, and turn allocation is performed by arranging the assigned component numbers. Machine allocation consists of several turns. Also, the sequence of component numbers within a turn indicate the nozzle picking them up. Figure 7 shows an example of nozzle allocation in component placement machines with four nozzles.

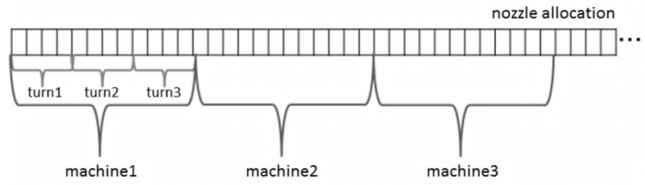


Figure7 Nozzle allocation in component placement machines

3.2 Determination of component picking-up and placing order

To evaluate a solution corresponding to a certain nozzle allocation, the component pick-up order and the component placing order are determined as described below, and the respective head travel distances and number of pick-ups are calculated.

3.2.1 Determination of the order of component picking-up order

When the nozzle allocation is given, the position of the head above the component feeder is settled. It is possible to pick up components in the shortest head travel distance by picking them up in order from the left or right of the head position. Also, the head position during pick-up may overlap for several components and, in this case, the components can be picked up at the same time with one up-down motion (gang pick mentioned before).

3.2.2 Determination of component placing order

When the nozzle allocation is given, the center of the head above the circuit board when placing each component, as shown in Figure 8, is settled. Here, the shortest path when traveling around the head centers can be regarded as a type of traveling salesman problem (TSP). Therefore, the placing order is determined using the nearest neighbor (NN) method, which is often used as the initial solution to the TSP. The NN method is a typical greedy method, and it generates a solution by repeating the operation of moving from the current point to the nearest point until all points have been visited.

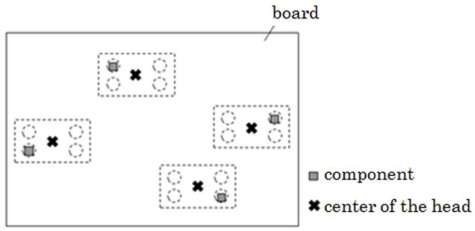


Figure8 Location of the center of the head

3.3 Improvement of component placing order

After component nozzle allocation has been improved, improvement of the placing path of each turn is carried out. A local search technique based on the first admissible move strategy using a 2-opt neighborhood is used to improve the placing path. A 2-opt neighborhood is a set of solutions generated by swapping two branches on a path, and it is often used to solve problems such as the TSP.

4 Computational experiment

A computational experiment was carried out to evaluate the effectiveness of the proposed method. The experiment environment consisted of an Intel Pentium CPU B940 2.00 GHz with 4.00 GB of memory, and the language used was C.

The number of slots in the component feeder was set at 30 and the distance between slots at 10 mm. The positional relationship of the circuit board and the component feeder and camera is set as shown in Figure 9. Also, the number of

nozzles was 16, and these nozzles were arranged on the head as shown in Figure 10. These settings were determined with reference to real component placement machines.

The experiment was done for 27 types of circuit boards randomly generated.

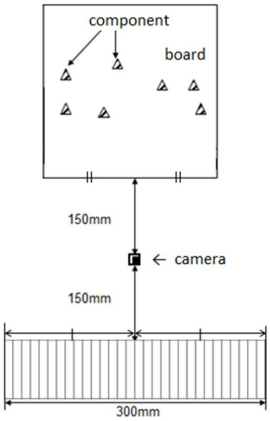
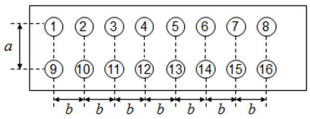


Figure9 Location of the circuit board and the component feeder and camera



$a = b = 10$mm

Figure10 Location of the nozzles on the head

4.1 Comparison with the conventional methods in the case of a single machine

An experiment comparing the proposed algorithm and conventional methods was carried out with regard to optimization of the pick-up and placing actions of one machine.

As for conventional methods, we tried two types of methods; pick-up first and placing first. In the former (pick-up first) method, picking-up action, turn allocation and nozzle allocation are determined before placing, which is determined by NN and 2-opt methods. This method reflects the actual process of optimization in the production of electronic circuit boards in industry. In the latter (placing first) method, the order of placing, turn allocaton and nozzle allocation are fully optimized before picking-up. Although this latter method is rather imaginary and not used in real industry, we conceived it in order to explore the superiority of picking-up first and placing first.

Figure 11 shows the relationship between the computation time and the evaluation value for the proposed algorithm and the conventional method (both 2 types) with regard to Data 14. It can be seen that the proposed algorithm converges to a better solution than those by the conventional methods.

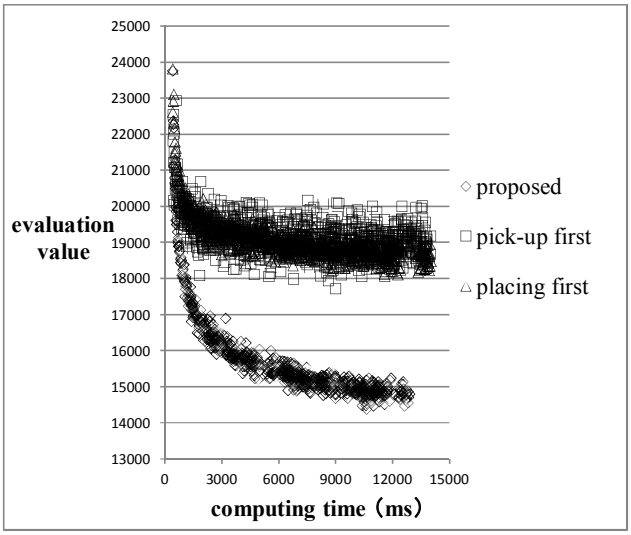


Figure11 Relationship between the computation time and the evaluation value (Data 14)

Table 1 gives the evaluation values obtained after a sufficient length of time using the algorithm of the conventional methods and the evaluation values obtained after an equivalent length of time using the proposed algorithm. The algorithm of the proposed method gives better solutions for all data than those by the algorithms of the conventional method.

As for the conventional method with pick-up first, Table 1 shows that the gap increases as the circuit board size increases. This is presumed to be because the cost of the placing action increases in proportion to the circuit board size, and so the proposed method, in which searching progresses while taking account of placing, becomes more advantageous. Table 1 shows also that the gap decreases as the number of component types increases. This is presumed to be because the greater the number of component types, the greater the cost of the pick-up action, and so improvement progresses easily, even in the conventional method.

As for the conventional method with placing first, Table 1 shows that the gap decreases as the circuit board size increases. This is presumed to be because the cost of the picking up action becomes more significant than the placing action as the circuit board size increases.

Table 2 gives the evaluation values obtained by the proposed method and the modified version of the proposed method where SA (simulated annealing) is used instead of local search with sufficient length of time. Parameters of SA have been adequately by preparatory test. Table 2 shows that local search gives relatively good solution in short time of computation, although the evaluated value itself is rather inferior to SA. In addition, the gap between SA and local search grows as the number of components increases; 3.71% for 64 components, 5.96% for 256 components, 13.98% for 512 components, and so on. This is presumed to be because when the number of components increases, the search space grows wide and LS tends to fall into a local optimal solution.

Table1 Comparison of conventional procedure and our algorithm

	Data details			pick-up first (A)		placement first (B)		proposed		eval gap A-proposed (%)	eval gap B-proposed (%)
Data No.	chip num	variety of chips	board size (mm)	eval	time (ms)	eval	time (ms)	eval	time (ms)		
1	64	10	100	3279	3621	3860	3633	2912	3486	11.19	24.56
2	64	10	200	4496	3590	4403	3552	3615	3445	19.60	17.90
3	64	10	300	5739	3585	5038	3571	4789	3482	16.55	4.94
4	64	20	100	3305	3551	4144	3593	3042	3450	7.96	26.59
5	64	20	200	4436	3553	4538	3663	3803	3428	14.27	16.20
6	64	20	300	5368	3538	5289	3584	4594	3454	14.42	13.14
7	64	30	100	3218	3545	4022	3731	3089	3409	4.01	23.20
8	64	30	200	4341	3550	4773	3594	3849	3426	11.33	19.36
9	64	30	300	5428	3578	5352	3596	4857	3416	10.52	9.25
10	256	10	100	13250	15162	15343	15382	11149	13765	15.86	27.33
11	256	10	200	17711	15108	18227	15269	14387	13771	18.77	21.07
12	256	10	300	22936	14906	20221	15191	17507	13750	23.67	13.42
13	256	20	100	12976	15057	16006	15227	11321	13739	12.75	29.27
14	256	20	200	17290	15150	18754	15348	14620	13703	15.44	22.04
15	256	20	300	23071	15077	21562	15198	18597	13783	19.39	13.75
16	256	30	100	12821	14990	16651	15430	11667	13399	9.00	29.93
17	256	30	200	18326	15040	19356	15327	14798	13515	19.25	23.55
18	256	30	300	22594	15250	20998	15172	18050	13414	20.11	14.04
19	512	10	100	25547	33652	31170	34465	22592	28267	11.57	27.52
20	512	10	200	35156	33541	36402	34650	29271	28339	16.74	19.59
21	512	10	300	46162	33597	41730	34815	35168	28471	23.82	15.72
22	512	20	100	25928	33615	32683	34447	23648	28417	8.79	27.64
23	512	20	200	35660	33793	38029	34775	30260	28362	15.14	20.43
24	512	20	300	46414	34120	43257	34792	37356	28401	19.52	13.64
25	512	30	100	26390	34966	33271	34690	24313	27824	7.87	26.92
26	512	30	200	35682	34083	38552	34977	30995	27958	13.14	19.60
27	512	30	300	46360	33938	43797	34731	37784	27672	18.50	13.73

Table2 Comparison of LS and SA

	LS		SA		eval gap LS-SA (%)
Data No.	eval	time (s)	eval	time (s)	
1	2912	3.49	2789	695.05	4.22
2	3615	3.45	3583	633.20	0.89
3	4789	3.48	4646	626.12	2.99
4	3042	3.45	2935	623.42	3.52
5	3803	3.43	3686	681.97	3.08
6	4594	3.45	4348	613.21	5.35
7	3089	3.41	2893	623.50	6.35
8	3849	3.43	3790	757.92	1.53
9	4857	3.42	4590	627.47	5.50
10	11149	13.77	10678	2920.29	4.22
11	14387	13.77	13085	2870.90	9.05
12	17507	13.75	16065	2855.21	8.24
13	11321	13.74	10650	2942.82	5.93
14	14620	13.70	13659	2892.42	6.57
15	18597	13.78	17375	2937.27	6.57
16	11667	13.40	11070	3018.16	5.12
17	14798	13.52	14551	3102.37	1.67
18	18050	13.41	16918	2891.23	6.27
19	22592	28.27	19440	6400.85	13.95
20	29271	28.34	24483	6408.74	16.36
21	35168	28.47	30143	6286.78	14.29
22	23648	28.42	20648	6363.88	12.69
23	30260	28.36	25890	6383.37	14.44
24	37356	28.40	31791	6386.61	14.90
25	24313	27.82	21273	6323.90	12.50
26	30995	27.96	26685	6473.63	13.91
27	37784	27.67	32959	6417.25	12.77

4.2 Line balancing

An example of line balancing using the proposed algorithm is given. Figure 12 shows the relationship between the head travel distance and the computation time when placing is done by 1, 2, 4, and 8 component placement machines using Data 14. Under the computation time within 2 seconds, multiple searches were performed while gradually increasing the computation time. By using the algorithm of the proposed method, it can be seen that the evaluation value improves in all of the cases using different numbers of machines. Also, the rate of improvement from the respective initial solution is 27.8% for 1 machine, 27.7% for 2 machines, 25.0% for 4 machines, and 26.5% for 8 machines.

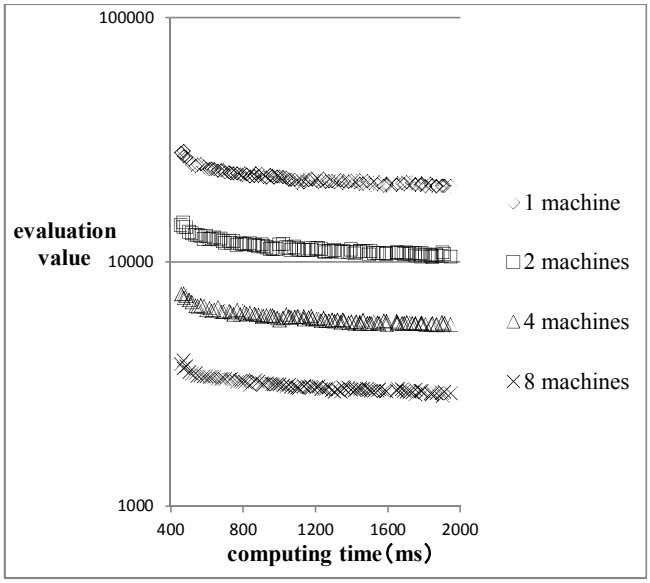


Figure12 Line-balancing of 1, 2, 4, and 8 component placement machines (Data 14)

5 Conclusions

We have discussed in this paper the motion optimization of multifunction component placement machines and proposed a model for comprehensive optimization of machine allocation, turn allocation, nozzle allocation, placing order, and pick-up order. Also, it has been confirmed by computer experiment that performing motion optimization using this model gives a sufficiently good solution compared to that of a conventional method.

Future work includes new algorithm of obtaining good solution as SA in a short time of computation even if the number of components increases. Also, it is desired to formulate the problem as an integer programming problem and obtain an exact optimal solution by IP solver and compare with the proposed method. Finally, it is hoped to create a model that can also handle cases in which some constraints for nozzle allocation is imposed, cases that also take the arrangement of components in the component feeder into account, and cases in which several types of circuit boards are produced.

References

[1] Mari Ayob, Graham Kendall, "A survey of surface mount device placement machine optimization: Machine classification," European Journal of Operational Research 186 (2008), 893–914

[2] Keisuke Yamamoto, Hidenori Ohta, and Mario Nakamori, "A Heuristic Algorithm for the Component Mounting Order Problem Based on Nozzle Allocation in Component Mounting Machines," IPSJ SIG Report, 2011-MPS-82(5), (2011), 1-6

[3] Hiroshige Tozaki, Hidenori Ohta, and Mario Nakamori, "A Heuristic Line Balancing Algorithm Accounting for Component Mounting Order," Proc. PDPTA 2011 (held in Las Vegas, Nevada, USA, July 11, 2011), 807-812.

Smart Home Delay Tolerant Network for an Earthquake Disaster

Raito Matsuzaki[1] **and Hiroyuki Ebara**[1]
[1]Kansai University, Yamate, Suita, Osaka, Japan

Abstract— *In a previous paper, we proposed a rescue support system for victims buried in an earthquake disaster by constructing an ad-hoc network using home-server smart homes. However, this system has the following two problems: i) it cannot ensure sufficient density of home servers for a WLAN communication range, ii) the system does not consider areas in which home servers cannot be used such as in parks and factories, for example. In this research, we propose a new method using a delay tolerant network (DTN) technique. In this method, rescuers with mobile devices relay information between disconnected networks by walking around during rescue activities. For a performance evaluation, we performed simulation experiments using a map of Abeno-ku, Osaka. From our results, we show that the proposed method increases the information acquisition rate, and the network can be maintained. In addition, we quantitatively show the penetration of the smart home needed for our system.*

Keywords: *ad-hoc* network, delay tolerant network(DTN), smart home, emergency rescue, earthquake

1. Introduction

In Japan, earthquake disasters are frequent. Problems during an earthquake disaster include communication blackouts and damaged infrastructures which cause traffic jams due to building collapse. Because of these problems, security companies and public institutions such as the fire department, often experience delays in their rescue activities. In the Great Han-Shin Awaji Earthquake, the survival rate within 24 hours was about 75 %, but only 15 % in 72 hours [1]. Making quick rescue activities is essential because the survival rate decreases with the passage of time.

In [2], we proposed a smart home network built with *ad-hoc* networks using home servers for smart homes, and a rescue support system for buried victims by making a rescue request map using this network in an earthquake disaster. The smart home manages various systems in the same way as home servers realizing a comfortable home environment through sensing. In the smart home network, home servers contain a wireless LAN (WLAN) and battery functions. Thus, this network will work even in the case of infrastructure blackouts and outages in the area because each home server forms an *ad-hoc* WLAN network. To use this network, a home server detects the presence of buried victims by information of their presence in collapsed homes and buildings, and sends neighboring home servers this information. Each home server makes a rescue request map based on this information. The rescue request map displays information of victims on the map. In addition, rescuers get this map from near home servers from rescuers with mobile devices. Therefore, we can identify the position of victims and rescue them quickly.

The smart home networking system (we previously devised), however, has a few problems. First, we have to ensure sufficient density of home servers for the WLAN communication range, and introducing this service into rural area is difficult. Therefore, this system is useful only in urban residential areas, and then only with a low penetration. Second, we consider places such as parks, factories, schools, and wide load trucks, not available to home servers. Thus, we proposed to set repeaters to aid communication between home servers in [2], but such repeaters are highly expensive.

In this research, we propose mobile device rescuers that can bring relay information to communicate between disconnected home servers. Mobile devices are introduced as a data exchange method in a delay tolerant network (DTN). The DTN communication system can transfer data with a long delay, but with some confidence even in environments of interruption and disconnection for communication during an earthquake disaster. The 'carry and forward' technique commonly used with DTN can exchange information between nodes. In a smart home network, this technique can relay information among disconnected home server groups because rescuers with mobile devices walk around and obtain information during a rescue activity. In this way, we can solve the problem of communication between disconnected home servers without the repeaters.

2. Delay Tolerant Network

In Epidemic Routing for Partially-Connected *Ad Hoc* Networks [3], techniques have been developed to allow message delivery in cases where a connected path from a source to a destination is not available in mobile *ad-hoc* networks. The authors show that Epidemic Routing achieves eventual delivery of 100 % of messages with reasonable aggregate resource consumption through an implementation in the Monarch simulator. Epidemic routing is generally used with DTN routing such as [4].

Delay Tolerant Networks (DTNs) [5] [6] are designed to overcome limitations in connectivity due to conditions such as mobility, poor infrastructure, and short-range radios. In fact, there are experimental projects such as the TIER [7] and

the KioskNet [8] due to internet connection in sparsely populated areas without communicational infrastructures. However, missed contact opportunities among nodes decrease throughput and increase delay in the network. By using capacity Enhancement with Throwboxes in DTNs [9], the authors of this article have proposed the use of Throwboxes to enhance network capacity in mobile DTNs. Furthermore, this capacity increased the transmission opportunities as well as throughput between nodes. The authors performed an extensive evaluation of their algorithms by varying both the underlying routing and the mobility models.

One type of research about the DTN routing technique is Data Routing for DTN Environments According to Data Size and Deadline [10]. In this research, the authors propose a technique to enable asynchronous communication between two points that cannot be used for infrastructures by the transport of data using a mobile device with which the user can move around. This proposed method uses the carry and forward techniques. In [10] the authors assume that each node introduces a small battery-powered server called an Infobox. Thus, they also assume that the Infobox and the mobile device are narrow-area wireless communication functions (WiFi, Bluetooth etc.), with enough storage, computing power, and memory to run the DTN. The novelty of this study [10], is not only improvement in the data arrival rate and shortening of the communication delay, but also consideration of the data size and the transmission deadline. Because the time a user has to communicate between nodes is limited in a real environment, users are not able to exchange all the data in time. Therefore mobile devices change the priority of the data to be sent by importance of data. This proposed method [10] can send and receive the higher value data and improve performance. In this proposed method, the authors performed simulation experiments in the Manhattan area. They show that the data arrival rate that considers the importance of data is higher than in other methods.

3. Smart Home Network with Delay Tolerant

In this section we give an overview of the smart home network and rescue support system, and propose a new method to improve connection in the smart home network. For more information about the smart home network and rescue support system, see [2].

3.1 Overview of Smart Home Network

In the smart home network, the home server operates a smart home system in the case of normal (non-emergency situations), and constructs *ad-hoc* networks by wireless functions between neighboring home servers in the case of a disaster. The smart home system shown Figure 1 represents a system that realizes a secure home environment in terms of monitoring, consumer electronics operations by sensing, and security.

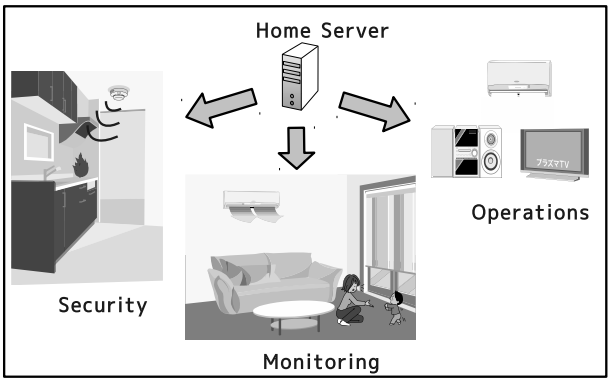


Fig. 1: Example of services by the smart home.

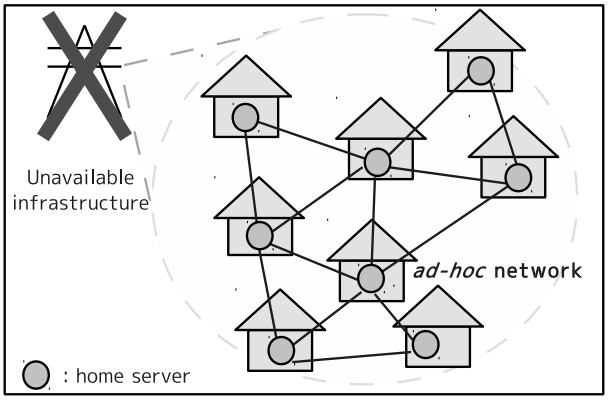


Fig. 2: Construction of *Ad-hoc* networks.

The smart home network uses Earthquake Early Warning (EEW) that exists in 2 types [11]. One type is for advanced users and another is for the general public. We use both EEWs in the smart home network, and we call the EEW for advanced users the first EEW, and the second type for the general public, the second EEW.

We describe the behavior the smart home network in an earthquake disaster. Receiving the first EEW, each home server checks connections between neighboring home servers. Receiving the second EEW, each home server sends and shares information (such as home rescue etc.) using infrastructures such as the Internet. When a blackout occurs after an earthquake, each home server connects and communicates with neighboring home servers to maintain the system, as shown in Figure 2. Thus, each home server needs a battery to run the system while electricity supply stops.

3.2 Overview of the Rescue Support System

It is very important for rescuers to rescue buried victims in an earthquake. However, such rescue operations to identify

Table 1: The amount of data needed to create a rescue request map.

information	amounts [bit]
rescue request information	1
home information	4
information of various sensors	7
complete rescue information	1
coordinate information	51

the location of buried victims are often difficult. In fact, the way rescuers now often find victims is by hearing the rescuers when they call for help. In this research, home servers create a rescue request map to quickly identify the location of buried victims.

The type of information needed to create a rescue request map is as follows:

- Rescue request information - a direct rescue request
- Home information - the number of people in a home
- Information of various sensors - the sensing signals of several sensors in the smart home system
- Information of disconnected home servers - home servers disconnected from the network by some abnormal reason emergencies, for example
- Complete rescue information - information to be sent by mobile devices after the rescuer rescues the victim
- Coordinate information - positional information of the home server

Deciding the amount of data for each piece of information needed to create a rescue request map is necessary. In Table 1 we show the amount of data needed to create a rescue request map. Rescue request information and complete rescue information need 1 bit (switching ON/OFF); home information needs 4 bits because the number of people in home is usually less than 15. The information about various sensors requires up to 7 bits, which can be used for up to 7 sensors. Coordinate information requires 51 bits because of GPS information. The longitude is 25 bits, and the latitude is 26 bits in the coordinate information. From Table 1, we assume the information required to create a rescue request map that needs at the minimum 8 Bytes.

We describe next how a home server creates a rescue request map. Home servers send information to create a rescue request map to each other, and share this information. Based on this information, each home server creates a rescue request map, and checks connections between neighboring home servers. If a home server updates information or receives new information, the home server sends it to a neighboring home server. The home server doesn't send duplicated information. The rescuer with mobile devices can get a rescue request map from the home servers, as shown in Figure 3.

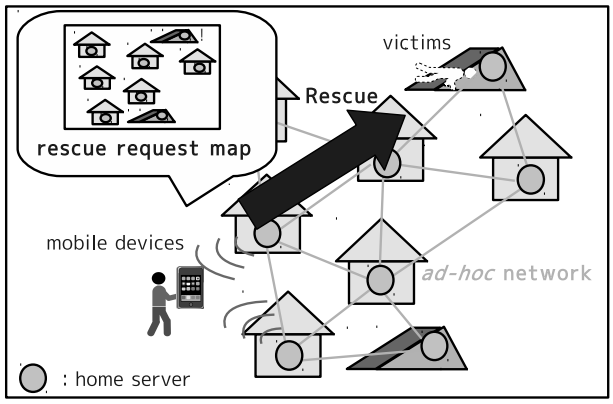


Fig. 3: Getting a rescue request map.

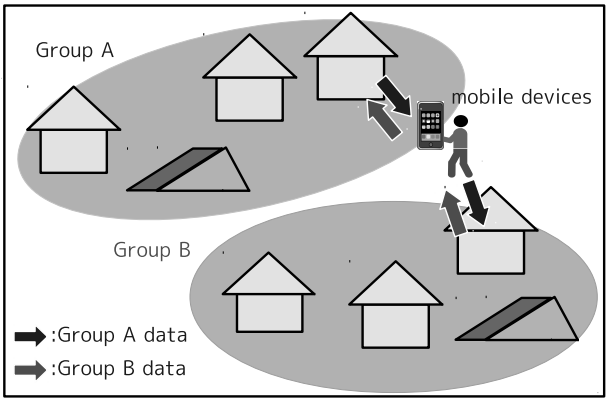


Fig. 4: Example of the mobile device relay.

3.3 Mobile Device Relay

To achieve the smart home network, we should consider two major problems. First, because the communication range of WLAN is small, the home server cannot connect to neighboring home servers. Second, cities have large areas such as mentioned before in which the home server cannot be used. Not putting a home server in these areas, causes several small home server groups. These problems cause a reduction in the connection rate if the smart home network is introduced. Therefore, we propose the mobile device relay to communicate between disconnected home servers in the smart home network, as shown Figure 4. Mobile devices have a data exchange method used in DTN, called a 'carry and forward' technique, so mobile devices can save information to create a rescue request MAP and pass it to other home servers. If a rescuer with mobile devices gets information to create a rescue request map from one home server group and moves to another home server group, the mobile devices can communicate with the home server group and share information between home server groups.

We propose an algorithm for sending and receiving information of a rescue request map for the proposed method as

follows:

Step1) When the rescuer with a mobile device moves with in the communication range of a home server, the mobile device sends rescue request map information to the home server.

Step2) If information the home server receives is new, the home server obtains this information.

Step3) If the home server has information the mobile devices do not have, the home server sends this information to the mobile device.

3.4 Assumptions necessary for Achieving the Proposed Method

To achieve the proposed method, it is necessary to consider several assumptions, described below.

3.4.1 Home Server

The home server has several functions in the smart home network as follows:

- Each home server has WLAN (IEEE 802.11) communications.
- Each home server can save all information to create a rescue request map.
- Each home server sends the differences in information to create a rescue request map.
- Each home server broadcasts between home servers, and broadcasts to mobile devices.

3.4.2 Mobile device

The mobile device of a rescuer has several functions as follows:

- Each mobile device has WLAN (IEEE 802.11) communications.
- Each mobile device gets information to create a rescue request map once.
- Each mobile device broadcasts to home servers (No transportable mobile devices).

The carry and forward technique needs to set a transmission deadline to avoid network congestion and waste of bandwidth. Nonetheless, rescuers with mobile devices do not discard information to create a rescue request map during the rescue activity. In addition, the size of information to create a rescue request map is small because it is less than the multiplication value of 8 Bytes by the number of home servers sharing information. Therefore, we do not need to worry about the transmission deadline for information to create a rescue request map, but only need to update new information sent to the home server.

In our proposed method, the mobile device can communicate with home servers, but cannot communicate with other mobile devices, called message passing communication. Using passing communication, mobile devices must always have an active communication function, which wastes the battery. Therefore, we assume that the mobile device only communicates with home servers.

Table 2: Parameters in simulations.

items	parameters
simulation area	Osaka Abeno-ku
simulation range	1 km^2
communication protocol	WLAN (IEEE 802.11)
communication range	15 m and 30 m
the number of households	8000
simulation time	15 minutes - 2 hours
measurement range (arrival rate)	300 m

3.4.3 Battery

When a large earthquake occurs, electricity supply is stopped because of damage to infrastructure functioning. Because the home server needs to supply it's own power, the home server has a large battery, or private power generation such as solar power. In addition, the home server is required to work for a minimum of three days. The sensor in the smart home and in the mobile device also needs to have enough battery to last for three days. We think, however, that battery capacity will increase through future technological development. Thus, we assume home servers, sensors, and mobile devices will have enough power supply to achieve the proposed method.

3.4.4 Movement of Rescuer

We assume rescuers move anywhere in areas in our simulation. Rescuers move freely in areas when mobile devices do not get the position of the victim. In our research, we assume that the free action of the rescuer follows a self-avoiding random walk [12] [13]. However, we propose that rescuers move a random distance up to one-tenth of the map in the same direction; otherwise, the random walk would be stuck in one place. When the mobile device gets the position of the victim, a rescuer can move there. If rescuers finish the rescue by finding the victims, then the rescuers can move freely again.

3.4.5 Communication Range of Home Server and Mobile Device

In our research, we assume that the communication range of the home servers and mobile devices is the same. We also assume that the communication range is equal in a circle. We assume that the communication range for WLAN is up to 30 m because we measured this range in [2].

4. Performance Evaluation

To show the effectiveness of the proposed method, we performed simulation experiments on an actual map. The simulator is implemented in Java.

Fig. 5: Map of simulation area in Abeno-ku.

4.1 Setting of Simulations

Table 2 shows the parameters in our simulations. We assume the simulation model is the residential area of Abeno-ku, Osaka, and the simulation area is within the range of 1 km square. The map of the simulation area is from the Geographical Survey Institute (GSI) [14]. The location of the home servers and the communication range are with respect to pixels[1]. The number of households in the simulation area is obtained from the number of households divided by the area of Abeno-ku. The penetration rate is obtained from the number of home server installations divided by the number of households in the simulation area.

Figure 5 shows the actual map of the simulations. The map is divided in monochrome by residential areas and not residential areas (home servers can be installed in the white spaces, but not in the black). The black areas represent places such as parks and factories as mentioned previously.

We simulate the information acquisition rate each of home servers and mobile devices in a simulation area. The information acquisition rate shows the rate of the information which the home server can get from others. We also simulate the arrival rate of a home server information in this area. This rate is the information acquisition rate measured the certain range with a focus on a home server, and this is because home servers should communicate in a narrow range to avoid large amount of information. We assume the certain range (the measurement range) is 300 m from preliminary experiments, as shown Table 2. Note that both the communication range and the measurement range are radiuses of a circle. If a mobile device communicates between disconnected home servers, both of the rates are added. In the simulation, the information acquisition rate for one home server (a single rate) is obtained from the number of other home server information divided by the number of households. The arrival rate for a single rate is obtained from the number of other home server information divided by the number of home servers in the measurement range. Then, we obtain the information acquisition rate and the arrival rate by taking average of a single rate in all home servers.

[1]Range of 1 pixel is 1.2 m (speed of people).

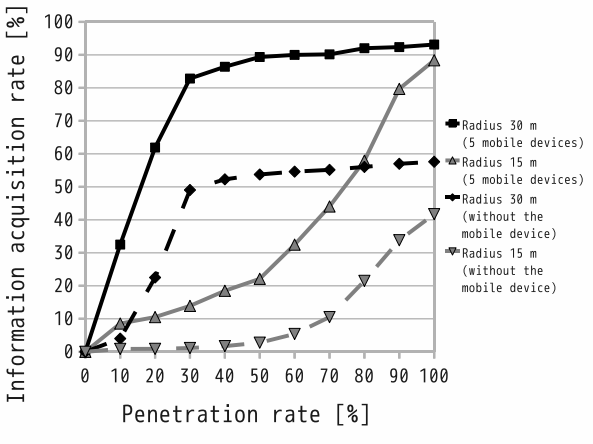


Fig. 6: Characteristics of the information acquisition rate for the home servers.

We do the simulation a hundred times, and then take the average.

4.2 Results of Simulations

We describe the simulation results about the information acquisition rate for home servers and mobile devices. Moreover, we also describe the result about the arrival rate of the home server information.

Figure 6 shows the information acquisition rate for home servers in the smart home network. The horizontal axis represents the penetration rate, and the vertical axis the information acquisition rate. The result of no mobile devices measures only the home server, which is the connection rate (the connection rate is obtained from the maximum number of home servers connected to each other divided by the number of households). The result of 5 mobile devices is due to the proposed method. In the proposed method, the number of rescuers is 5 and simulation time is 1 hour. From Figure 6, we can see that the information acquisition rate (the connection rate) without the mobile device for both 15 m and 30 m was less than 60 % even if the penetration rate was 100 %. Because the area in which home servers cannot be put is too large, several small home server groups are needed for this area. However, the information acquisition rate of the proposed method (with 5 mobile devices) is significantly improved compared to without the mobile device. Moreover, this system needs about a 50 % penetration rate to get over a 90 % connection rate in a communication range of 30 m for each 1 hour.

Figure 7 and Figure 8 shows the information acquisition rate of the mobile device relay for both home servers and mobile devices in the smart home network. The horizontal axis represents the simulation time, and the vertical axis the information acquisition rate. We simulate the penetration rate of the home server as 50 % and the communication range

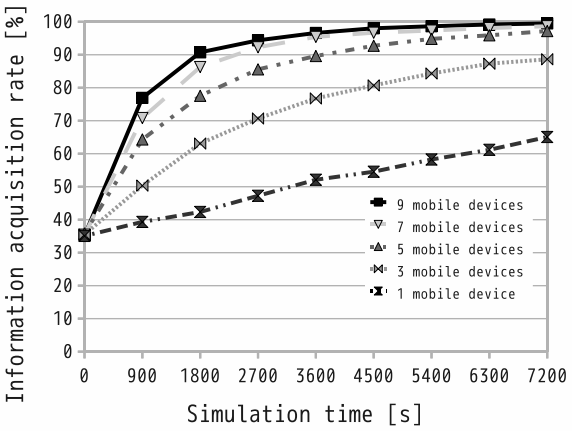


Fig. 7: Characteristics of the information acquisition rate for home servers by the simulation time.

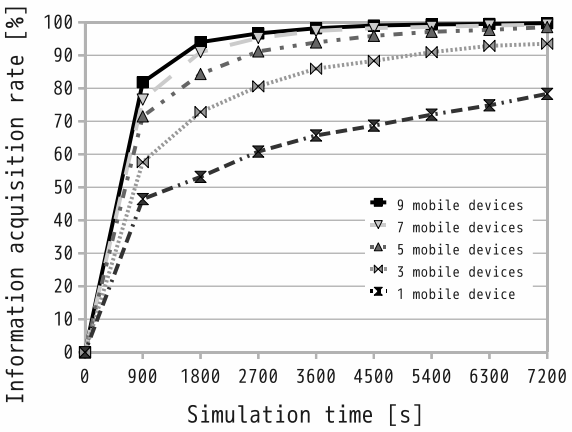


Fig. 8: Characteristics of the information acquisition rate for mobile devices by the simulation time.

as 30 m, and simulation time is 2 hour. From both Figure 7 and Figure 8, the more the simulation time increases, the more information acquisition rates increase. Especially, the result of both 7 and 9 mobile devices is over a 95 % within 1 hour, we consider this value is enough to maintain the smart home network.

Figure 9 shows the arrival rate for the home server information in the certain range with a focus on a home server as 300 m. The horizontal axis represents the simulation time, and the vertical axis the arrival rate. We simulate the penetration rate of the home server as 50 %, and simulation time is 1 hour. From Figure 9, the more the simulation time increases, the more arrival rates approach 100 % in a communication range of 30 m within 1 hour. Moreover, the larger the number of mobile devices, the earlier arrival rates approach 100 %.

The connection rate, therefore, is improved by our proposed method especially with in the communication range of 30 m. In this range we showed a performance good enough to provide the system with connection rate of at least 90 %. We simulated the communication rate equally in a circle. However, we need to consider that the communication range isn't equal in a circle because of the difference in height of the terrain and the placement of obstacles in reality.

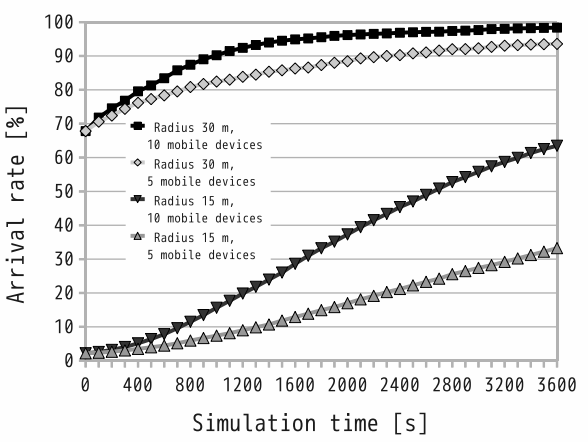


Fig. 9: Characteristics of the arrival rate for the home server information.

5. Conclusion

It is very important for victims to be rescued within 72 hours because this is generally how long victims can survive without help. To do this, rescuers must identify the location of buried victims. We proposed a rescue support system that creates a rescue request map. In our research, we proposed a method for a mobile device relay to communicate between disconnected home servers in the smart home network during an earthquake disaster. We showed that our proposed method is effective by performing an evaluation of the information acquisition rate.

We are currently implementing a communication environment to achieve a smart home network and a rescue support system.

Acknowledgement

This work was supported in part by Grant-in-Aid for Scientific Research(B) (23330098).

References

[1] Cabinet office, government of japan. (2012) Cabinet office, government of japan homepage on the document collection of precepts in the Great Hanshin Earthquake (in Japanese). [Online]. Available: http://www.bousai.go.jp/1info/kyoukun/hanshin_awaji/data/detail/1-1-2.html

[2] Raito Matsuzaki and Hiroyuki Ebara, "Ad-hoc networks using smart homes in an earthquake disaster, (in Japanese)" in *IPSJ SIG Notes 2011-MPS-83*, 2011, paper 2011-MPS-83(6), p. 1-7.

[3] A. Vahdat and D. Becker, "Epidemic routing for partially connected ad hoc networks," Duke University, Durham, Technical Report CS-200006, April 2000.
[4] Yong-Pyo KIM, Keisuke NAKANO, Kazuyuki MIYAKITA, Masakazu SENGOKU and Yong-Jin PARK, "A Routing Protocol for Considering the Time Variant Mobility Model in Delay Tolerant Network," *IEICE TRANSACTIONS on Information and Systems*, Vol.E95-D, No.2, pp. 451-461, February 2012.
[5] Forrest Warthman. (2003) Delay-Tolerant Networks (DTNs). [Online]. Available: http://www.ipnsig.org/reports/DTN_Tutorial11.pdf
[6] El Mastapha Sammou and Abdelmounaim Abdali, "Routing in Delay Tolerant Networks (DTN)" in *Int. J. Communications, Network and System Sciences*, Vol.4, No.1, pp. 53-58, January 2011.
[7] (2012) The Tier website. [Online]. Available: http://tier.cs.berkeley.edu/drupal
[8] (2012) KioskNet website. [Online]. Available: http://blizzard.cs.uwaterloo.ca/tetherless/index.php/KioskNet
[9] W. Zhao, Y. Chen, M. Ammar, M. Corner, B. Levine, and E. Zegura. "Capacity enhancement using throwboxes in DTNs," in *Mobile Ad-hoc and Sensor Systems (MASS). 2006 IEEE International Conference*, 2006, paper 10.1109/MOBHOC.2006.278570, p. 31-40.
[10] Weihua Sun, Yasuhiro Ishimaru, Keiichi Yasumoto and Minoru Ito, "Data routing for DTN environments according to data size and deadline, (in Japanese)" in *IPSJ SIG Notes 2010-MPS-80*, 2011, paper 2010-MPS-80(26), p. 1-6.
[11] Japan Meteorological Agency. (2012) Japan Meteorological Agency homepage on Earthquake early warnings. [Online]. Available: http://www.jma.go.jp/jma/en/Activities/eew.html
[12] Tracy Camp, Jeff Boleng and Vanessa Davies, "A survey of mobility models for ad hoc network research," *Wireless Communications and Mobile Computing*, Vol.2, Issue.5, pp. 483-502, August 2002.
[13] InKwan YU and Richard NEWMAN, "A Topology-Aware Random Walk," *IEICE TRANSACTIONS on Information and Systems*, Vol.E95-B, No.3, pp. 995-998, March 2012.
[14] Geospatial Information Authority of Japan. (2012) Geospatial Information Authority of Japan homepage on geographical survey institute (GSI). [Online]. Available: http://www.gsi.go.jp/ENGLISH/index.html

An Intelligent Lighting System Saving Power Consumption by Estimating Illuminance Sensor Positions

Mitsunori MIKI
Department of Science and Engineering
Doshisha University
Kyoto, Japan
mmiki@mail.doshisha.ac.jp

Takuro YOSHII
Graduate School of Engineering
Doshisha University
Kyoto, Japan
tyoshii@mikilab.doshisha.ac.jp

Keiko ONO
Department of Electoronics and Informatics
Ryukoku University
shiga, Japan
kono@rins.ryukoku.ac.jp

Yohei AZUMA
Graduate School of Engineering
Doshisha University
Kyoto, Japan
yazuma@mikilab.doshisha.ac.jp

Kazuki MATSUTANI
Graduate School of Engineering
Doshisha University
Kyoto, Japan
kmatsutani@mikilab.doshisha.ac.jp

***Abstract*—We are doing researching and development of an intelligent lighting system to provide desired brightness to a desire place. To realize individual lighting environments, this system needs the influence level of each lighting and each illuminance sensor. Also, by measuring the influence level that dynamically, this system can cope with dynamic illuminance sensor relocations (user relocations). If this system don't consider illuminance sensor relocations, this system can turn off lightings of little influence. Therefore, this system leads to energy saving. However, It is impossible to be successful at both illuminance sensor relocations and turning off lightings (improvement of energy efficiency). To solve this problem, this paper proposes a new lighting control algorithm which estimates illuminance sensor positions based on distances from turned-on lightings. It will enable the lighting system to learn the change in user or desk positions, and turn off those lightings which affect users very slightly. we conducted an experiment to simulate a real office. As a result, we indicate that the algorithm can realize the illuminance level desired by users while further saving energy consumption.**

I. Introduction

With the development of information technology and hardware, many electronic appliances such as televisions, air conditioners and fans, now incorporate an intelligent system which controls the operation of the appliance to suit environmental conditions, reducing the need of human operation.

Against such backdrop, intelligent controls are being introduced also for lighting systems from the aspect of saving energy consumption. One example of such systems is the "Energy-Saving Lighting Control System Linked to Access Control" by Mitsubishi Electric Corporation[1]. This system has infrared sensors built into lighting fixtures to detect user position coordinates and controls the luminous radiation of variable-light lighting fixtures. In this way, it can bring the light from the lighting system only to areas where a user is present and prevent excess light to save energy consumption. This system, however, uses an approach that controls each lighting fixture and does not consider providing a given brightness (illuminance) at a given point.

On the other hand, it has been reported that, to realize a better office environment, providing an illuminance optimized for each worker's task is effective for raising worker productivity[2]. One lighting solution which can realize individualized illuminance levels in an office environment is a task-ambient lighting system[3]; but in Japanese offices, task-ambient lightings are rarely adopted. This is because typical office buildings are equipped with ceiling lighting fixtures which provide a uniform illuminance, and most companies are unwilling to pay extra costs for adding task-ambient lightings. Considering these, there is a need of a lighting control system which provides light optimized for each office worker using only ceiling lighting fixtures.

Against this backdrop, the authors have proposed an intelligent lighting system which can provide brightness as required by users at any given points specified by users, depending only on ceiling lighting fixtures[4], [5]. The intelligent lighting system is composed of lighting fixtures, a lighting control devices, illuminance sensors (one person holds one illuminance sensor), and an electrical power meter. With this intelligent lighting system, each user specifies a target illuminance level for an individualized illuminance sensor placed on the desktop, then the system will follow a lighting pattern which realizes the target illuminance level while minimizing energy consumption using an optimization method. One should also note that this intelligent lighting system does not consider turning off part of lightings in the office to cope with dynamic changes in environmental conditions such as user relocations and change

in desk positions.

The intelligent lighting system has proven successful in our laboratory experiments[6]. Toward the commercialization of our intelligent lighting systems, currently verification experiments are underway in several offices in Tokyo and Fukuoka[7]. In the experiment, it was found that users hardly changed desk positions in these offices. Under such circumstances, we introduced a lighting control algorithm which turns off lights in places requiring no brightness to further improve the energy efficiency.

On the other hand, there are many "non-territorial" offices where users can move or change desk positions rather frequently. In view of this, this paper proposes a new lighting control algorithm which estimates illuminance sensor positions even after they are relocated. It will enable the system to cope with user relocations and desk position changes, while turning off those lightings which hardly affect users.

In this study, an operational experiment simulating a real office environment was conducted to verify the effectiveness of the proposed system.

II. INTELLIGENT LIGHTING SYSTEM

A. Construction of Intelligent Lighting System

An intelligent lighting system realizes an illuminance level desired by the user while minimizing energy consumption by changing the luminous intensity of lightings. The intelligent lighting system, as indicated in Fig.1, is composed of lights equipped with microprocessors, portable illuminance sensors, and electrical power meter, with each element connected via a network.

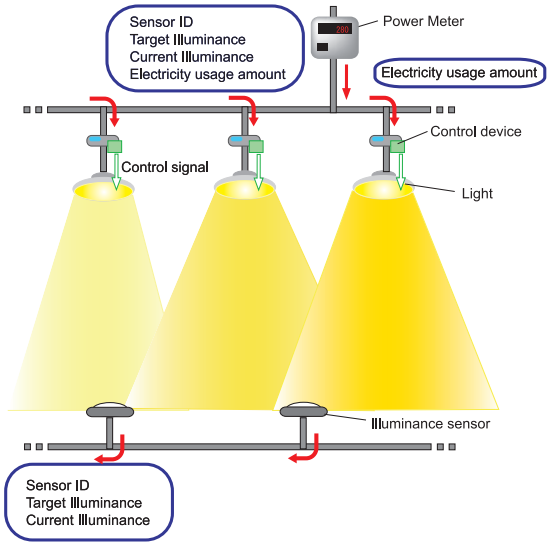


Fig. 1. Configuration of intelligent lighting system

B. Adaptive Neighborhood Algorithm using Regression Coefficient(ANA/RC)

The control algorithm is a critical element for the control of an intelligent lighting system. The speed of convergence to the target illuminance as well as its accuracy depends largely on the lighting control algorithm. As the best algorithm presently available for lighting control, we have proposed an Adaptive Neighborhood Algorithm using Regression Coefficient (ANA/RC)[8], which was developed by adapting the Stochastic Hill Climbing method (SHC) specifically for lighting control purposes.

In ANA/RC, the design variable is the luminous intensity of each lighting: the algorithm aims to minimize the power consumption while keeping the illuminance at the target level or above. It further enables the control system to learn the effect of each lighting on each illuminance sensor by regression analysis and, by changing the luminous intensity in response, enables a quick transition to the optimum intensity.

The following is the flow of control by ANA/RC:

1) Each lighting lights up by initial luminance.
2) Each illuminance sensor transmits illuminance information (current illuminance, target illuminance) to the network. The electrical power meter transmits power consumption information to the network.
3) Each lighting acquires the information from step 2), and conducts evaluation of objective function for current luminance.
4) Neighborhood is determined, which is the range of change in luminance based on factor of influence and illuminance information.
5) The next luminance within the neighborhood is randomly generated, and the lighting lights up by that luminance.
6) Each illuminance sensor transmits illuminance information to the network. The electrical power meter transmits power consumption information to the network.
7) Each light acquires the information from step 6), and conducts evaluation of objective function for next luminance.
8) The system performs regression analysis based on the luminous intensity data from each light and illuminance data from each illuminance sensor to determine the regression coefficient (influence level).
9) If the objective function value is improved, the next luminance is accepted. If this is not the case, the lighting returns to the original luminance.
10) If any of the lightings has been at the minimum lighting luminous intensity for a certain time with only a small influence level, the system turns it off (applicable only when there is no illuminance sensor relocation) .
11) Steps 2) through 10) constitutes one luminous intensity value search operation, which is repeated.

A search operation process (requiring about 2 seconds) consists of steps 2) through 10) above: by iterating this process, the system continues to learn how the lighting affects the illuminance sensor measurement until it realizes the target illuminance with minimum power consumption. Furthermore, by using the influence level found in step 8) as a basis for the evaluation and generation of the next illuminance value, the

system can quickly optimize illuminance.

Next, we will see the objective function used in this algorithm. The purpose of the intelligent lighting system is to achieve each user's desired illuminance, and to minimize energy consumption. Thus, it can be understood as an optimization problem in which each light optimizes its own luminance. Following from this, the luminance of each light is considered a design variable, under the constraint of the user's target illuminance, in resolving the problem of optimization to minimize energy consumption. For this reason, the objective function is set as in Eq. (1).

$$f = P + w\sum_{i=1}^{n} g_i \quad (1)$$

$$g_i = \begin{cases} (It_i - Ic_i)^2 & I_* \leq |It_i - Ic_i| \\ 0 & otherwise \end{cases} \quad (2)$$

P: Power consumption, w: Weight
Ic: Current illuminance, It: Target illuminance
n: Number of users
I^*: Threshold on illuminance difference

The objective function was derived from amount of electric power P and illuminance constraint g_j. Also, changing weighting factor w enables changes in the order of priority for electrical energy and illuminance constraint. The illuminance constraint is decided so that a difference between current illuminance and target illuminance within a threshold, as indicated by Eq. (2). The threshold value is set as a 50 lx.

Since this intelligent lighting system uses an autonomous distributed-control algorithm, particular cases of installation may use either distributed control or centralized control.

III. VERIFICATION EXPERIMENTS IN REAL OFFICE ENVIRONMENTS

From around 2009 onward, we have conducted experiments to verify the effectiveness of the intelligent lighting system in several offices in Tokyo and Fukuoka.

In an intelligent lighting system, to cope with user relocations and change in desk positions, even those lightings which hardly affect users need to be kept on at a minimum luminous intensity level. However, in the offices where our experiments were conducted, workers' desk positions were basically fixed and user relocations were rare. Hence, to further improve the system's energy efficiency, we have introduced a control mechanism which turns off lightings at points requiring no light <item (10) listed in 2.1> in these offices.

In this system, for each illuminance sensor, the IDs of several lightings around it are registered on a database, which are chosen based on the levels of influence of lightings to the sensor. By using the data, each lighting can be controlled optimally and turned off when appropriate. However, this method cannot cope with illuminance sensor relocations.

Meanwhile, not a few offices now use a non-territorial office design, in which workers have no fixed personal desks. In non-territorial offices, users may choose any desk position and user relocations are easy. If the control method which may turn off some lights is introduced in this type of office with a dramatically changing environment <item (10) listed in 2.1>, then the system will not be able to optimize lightings once a user relocation occurs, because it will be impossible to calculate the level of influence by a turned-off lighting onto the relevant illuminance sensor.

One solution to this problem may be to require users to notify the system of every user relocation: then the system turns on all lightings that have been turned off and calculate the influence levels of each lighting to relevant illuminance sensors and update the database. But in a real office, turning on all lights that have been off every time a worker relocates may disturb other workers in the office. In addition, users take time to notify the system that they relocate.

Hence, we propose a new lighting control algorithm for the system to automatically detect a user relocation and turn on only those lights which have a high level of influence over the relocated user.

IV. LIGHTING CONTROL ALGORITHM BASED ON ESTIMATED ILLUMINANCE SENSOR POSITIONS

A. Basic Idea of the Proposed Method

This study proposes a new lighting control algorithm which can cope with illuminance sensor relocations when some lightings have been turned off. In the study, regression analysis is performed based on the luminous intensity data from lightings which are on and the illuminance sensor data after the relocation, then the new sensor position is estimated based on their influence levels. After that, the lighting closest to the estimated sensor position is identified. If that lighting is off, the system can calculate the influence level by turning it on. This will enable the system to cope with user relocations while minimizing discomforts on other office workers. Moreover, the relocated user does not need to take the trouble of notifying the system of the relocation.

To realize this, the following control steps are added to the intelligent lighting system control algorithm:

1) The system detects an illuminance sensor relocation.
2) Regression coefficients for turned-on lightings and the relocated illuminance sensor are calculated.
3) Based on the regression coefficients obtained from step 2), the distances from turned-on lightings to the relocated sensor are calculated.
4) Based on the calculated distances, the position coordinates of the relocated illuminance sensor are estimated.
5) Based on the position coordinates of lightings and the estimated position coordinates of the relocated illuminance sensor, the system finds the lighting closest to the sensor and turns it on if it is off.

Illuminance sensor relocations are detected based on the amount of changes in the illuminance level. When a change in illuminance occurs which cannot be explained from changes

in the luminous intensities of adjacent lightings alone, the system assumes it to be a sensor relocation. Unlike our former intelligent lighting systems, position coordinates of each lighting need to be known because relocated sensor positions are estimated from its distances to lightings. Also, for each illuminance sensor, at least three of those lightings which have high level of influence onto the sensor should be on at any point of time. This is to enable the system to estimate the position of a relocated sensor based on the distances from those lightings which are on. Once the position coordinates of the relocated illuminance sensor are estimated, the number of lightings to be turned on is determined more than three.

Using the method described above, the system can turn off those lightings which hardly affect users while coping with user relocations and desk position changes.

B. Estimating Distances Based on Regression Coefficients

To estimate distances from turned-on lightings to a relocated illuminance sensor, regression analysis is used. Regression analysis is a method to derive a regression equation to explain the causality relation between change in the explanatory variable L and change in the observed value I, which is shown in this case by Eq. (3):

$$I = \sum_{i=1}^{n} r_i \times L_i + \beta \quad (3)$$

I: Illuminance, r: Regression coefficient
L: Luminous intensity, β: Constant
n: Number of lightings

In an intelligent lighting system, regression analysis is performed using explanatory variable L which is the luminous intensity of a lighting, and an observed value I which is the illuminance measured by an illuminance sensor, to calculate the regression coefficient r. The regression coefficient r, which explains the regression equation, is the level of influence by the lighting to the illuminance sensor. Based on this regression coefficient r, the distance between the lighting and the illuminance sensor is estimated.

The relation between luminous intensity L and illuminance I is expressed by Eq. (4) [9], where illuminance I is in inverse relation with the square of the distance d from the light source. Also, the relation between the distance d from the light source and the regression coefficient r expressed by Eq. (3) is expressed by Eq. (5).

$$I = \frac{L}{p^2} \quad (4)$$

$$\Leftrightarrow \quad d^2 = \frac{L}{I} = \frac{1}{r}$$

$$\Leftrightarrow \quad d = \frac{1}{\sqrt{r}} \quad (5)$$

r: Regression coefficient, I: Illuminance
L: Luminous intensity
d: Distance from the light source

Eq. (5) shown above indicates that the regression coefficient r in the intelligent lighting system indicates an estimation of distance d from the light source (distance between the lighting and the illuminance sensor).

Thus, an experiment was conducted to verify the distances between lightings and illuminance sensors and their relation with regression coefficients. In the experiment, 15 neutral white fluorescent lamps (Panasonic FHP45EN) and 4 illuminance sensors were installed as shown by Fig.2, and between each lighting and each illuminance sensor, the distance and the regression coefficient were calculated.

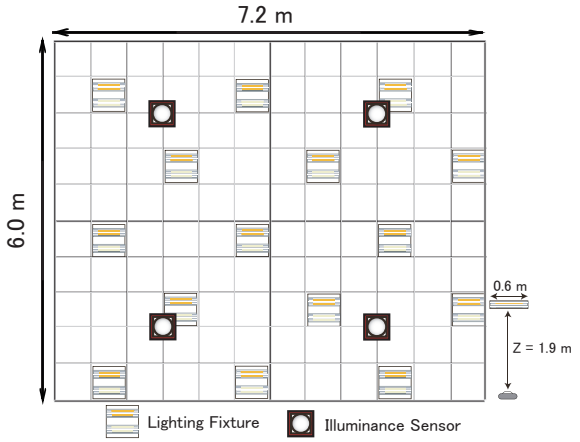


Fig. 2. Experimental environment to obtain regression coefficients

Fig.3 shows the distances between lightings and illuminance sensors and respective regression coefficients.

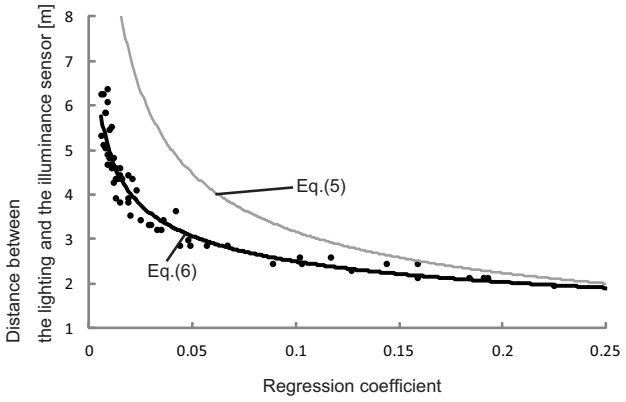


Fig. 3. Relation between the regression coefficient and distance

Fig.3 compares the regression coefficients obtained from the experiment were with the Eq. (5). The result indicates that when the distance between the lighting and illuminance sensor is shorter, the disparity is smaller; when the distance is larger, the disparity is greater. This is because Eq. (4) does not take account of the lighting fixture's radiation characteristics. Fig.4 shows the luminous intensity distribution curve curve for the fluorescent lamp (Panasonic FHP45EN) used in the experiment.

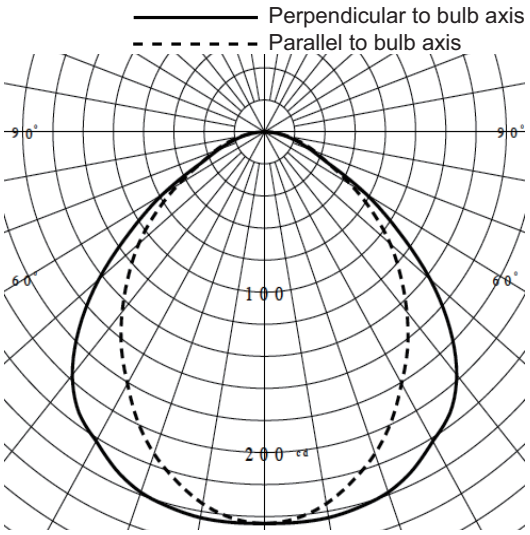


Fig. 4. Luminous intensity distribution curve(FHP45EN)

Luminous intensity distribution curve is a polar plot representing the luminous intensity as a function of angle about a light source. From Fig.4, one can see that the illuminance on the illuminated plane decrements more when the angle between the lighting fixture and the illuminated plane is larger. In the experimental results shown by Fig.3, the angle is larger when the distance between the lighting and the sensor is larger. Hence, the greater the distance is, the more largely the resulting regression coefficient falls below Eq. (5). Therefore, it is necessary to derive a model taking into account the lighting fixture's radiation characteristics.

In the proposed method, a mathematical model was derived, which estimates the distance based on the regression coefficient obtained through the experiment as shown by Fig.3 using the least square method. The derived mathematical model is shown by Eq. (6).

$$d = 1.8044 \times r^{-3.221} \tag{6}$$

r: Regression coefficient
d: Distance between the lighting and the illuminance sensor

Under this environment, the radiation characteristics of the lighting fixture is taken into account by using Eq. (6), to reduce the error in estimating the distance using the regression coefficient.

C. Estimating an Illuminance Sensor Position

Based on the method described in the section above, we can estimate the distance between a turned-on lighting and a relocated illuminance sensor. Based on the estimated distance, the position of the relocated illuminance sensor is estimated using Eq. (7) below.

$$d_i = \sqrt{(X - x_i)^2 + (Y - y_i)^2 + (Z - z_i)^2} \tag{7}$$

d_i: Distance between the turned-on lighting and the illuminance sensor
X, Y, Z: Position coordinates of the relocated illuminance sensor
x_i, y_i, z_i: Position coordinates of the turned-on lighting
i: Number of turned-on lightings

Now because the unknown values are the position coordinates (X, Y, Z) of the relocated illuminance sensor, a unique solution will be found by writing three or more equations. However, for a lighting that is turned off, it is impossible to estimate the distance because there is no way to calculate a regression coefficient concerning any illuminance sensor. Considering this, the system is designed to keep at least three lightings of relatively high influence levels always turned on per illuminance sensor to enable the estimation of position coordinates of relocated illuminance sensors.

Using the method described above, the position of a relocated illuminance sensor is estimated and three or more lights in its vicinity are turned on. This enables the system to calculate the regression coefficient for a lighting near the relocated illuminance sensor which had been turned off.

V. Operational Experiment

A. Overview of the Operational Experiment

To verify the effectiveness of the proposed method, an operational experiment was conducted for 30 minutes changing luminous intensity of the lighting in 2-second steps (900 steps). For the experiment, 15 lighting fixtures were installed as shown by Fig.5 to simulate a workplace with 3 users (using 3 illuminance sensors). For illuminance sensors A, B and C, the desired illuminance level was set at 300, 400 and 500 lx respectively. The lighting fixtures used were neutral white fluorescent lamps (Panasonic FHP45EN) which had a variable lighting luminous intensity range between a minimum of 30 % and a maximum of 100 %.

In the experiment, when the illuminance value is within the range between +6 % and -8 % of the desired level, the desired illuminance level is deemed to be achieved.

The operational experiment was conducted using the following two patterns to compare the system's performance with a conventional intelligent lighting system as well as to verify the energy saving effect of the new method.

- An operational experiment using our conventional lighting control algorithm (conventional method)
- An operational experiment using a control algorithm incorporating the proposed method (proposed method)

In the operational experiment, after 250 steps, the illuminance sensor A (at illuminance setting point A) is relocated to the new position shown by Fig.5. When an illuminance sensor relocation is detected using the proposed method, the relocated sensor position is estimated using illuminance and luminous intensity data over 60 steps.

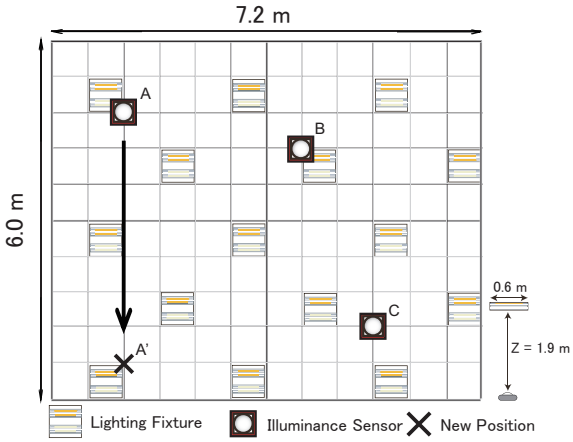


Fig. 5. Experimental environment

B. EXPERIMENT RESULTS AND DISCUSSIONS

We checked whether the system realized the desired illuminance level at each illuminance setting point in each of the conventional method and the proposed method. Fig.6 shows the historical illuminance levels from the conventional method and Fig.7 shows those from the proposed method.

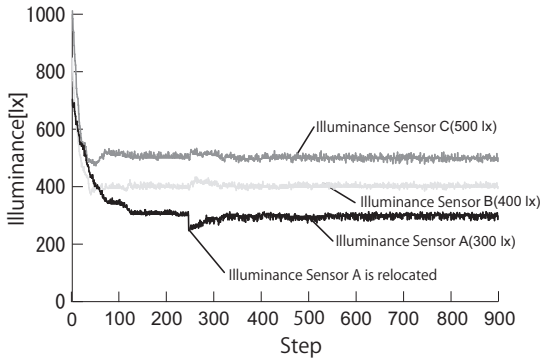


Fig. 6. History of the illuminance data (conventional method)

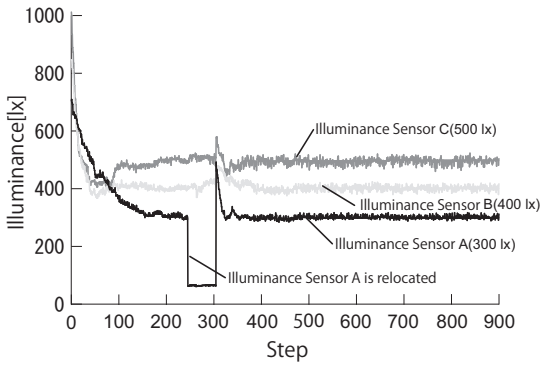
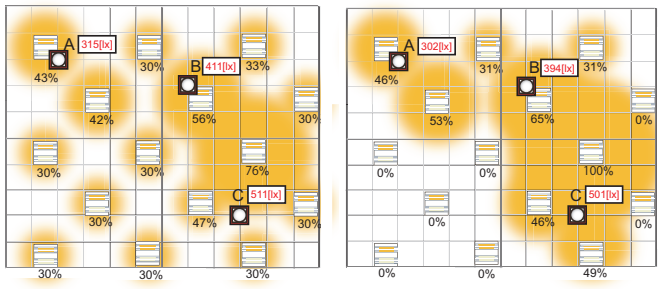


Fig. 7. History of the illuminance data (proposed method)

Fig.6 and Fig.7 show that in both methods the system realized desired illuminance level at each illuminance setting point. It is shown also that both methods successfully coped with the relocation of illuminance sensor A.

Next, to verify the energy saving effect, the status of each lightings before the sensor relocation at the point of 200th step is shown in Fig.8.



(a)Conventional method (b)Proposed method

Fig. 8. Status of lightings (at the point of 200th step)

From Fig.8-(a) showing the status of lightings in the conventional method, we can see that lightings near each illuminance setting point are turned on at a high luminous intensity level while those distant from an illuminance setting point are turned on at a low luminous intensity level. To cope with sensor relocations, however, all lightings need to be kept on at least at a low luminous intensity level. Meanwhile, from Fig.8-(b) showing the status of lightings in the proposed method, we can see that the lightings near each illuminance setting point are turned on at a high luminous intensity level while those distant from an illuminance setting point are turned off.

Then illuminance sensor A is relocated at the point of 250th step. In the proposed method, after detecting the relocation, the system calculates influence levels concerning turned-on lightings to estimate the new position. Fig.9 shows the estimated and actual position coordinates of relocated illuminance sensor A.

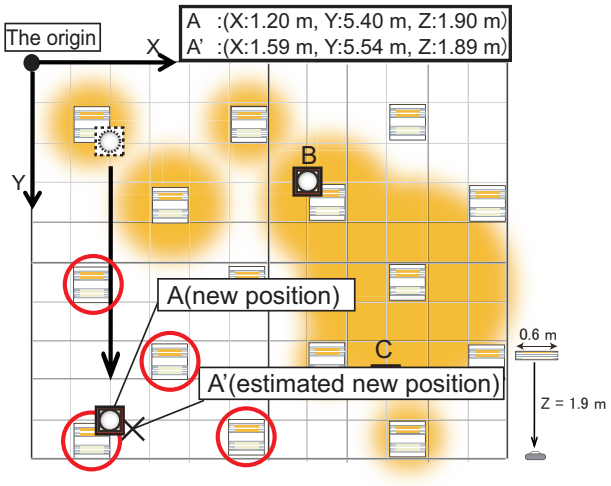
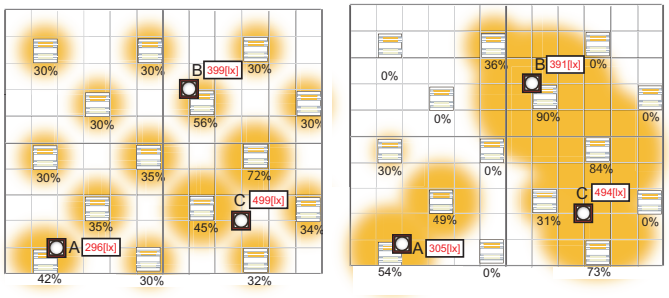


Fig. 9. Status of lightings at the point of 310th step (proposed method)

Fig.9 shows that the proposed method is able to estimate the position of illuminance sensor A after relocation. This means that by turning on the four lightings (lightings in circles shown in Fig.9 closest to the estimated sensor position, the regression coefficients for the lightings presumed to be most influential can be calculated.

Finally, Fig.10 shows the status of lightings after sensor relocation (at the point of 500th step).



(a)Conventional method (b)Proposed method

Fig. 10. Status of each lightings (at the point of 500th step)

Fig.10-(a) shows that in the conventional method, the energy consumption levels of some lightings were lowered after detecting a sensor relocation. But anyway, even lightings of little influence were not turned off but kept on. On the other hand, in the proposed method shown by Fig.10-(b), those lightings expected to have a high influence level are turned on and regression coefficients are calculated. Then based on the calculated regression coefficients, lightings of little influence are turned off to realize a greater energy saving effect than in the conventional method.

From the results of the experiment, it has been demonstrated that the proposed method can also realize the illuminance levels as desired by users while saving energy consumption.

C. Comparison of Power Consumption Results

This section compares the power consumption by the lightings between the conventional method and the proposed method. Fig.11 shows the historical records of power consumption by the lightings controlled by the conventional method and the proposed method.

From Fig.11, we can see that the proposed method consumes about 30 % less power than the conventional method. As a result, it has been demonstrated that the proposed method realizes a performance equivalent to the conventional method while further reducing energy consumption.

VI. CONCLUSION

Ever since 2009, we have been conducting verification experiments for our intelligent lighting systems at several offices in Fukuoka and Tokyo. Since most of these offices employ a fixed desk system, we introduced an algorithm which turns off lightings of smaller influence levels assuming no user relocations. However, in offices where users often move, once some lightings are turned off, it becomes impossible

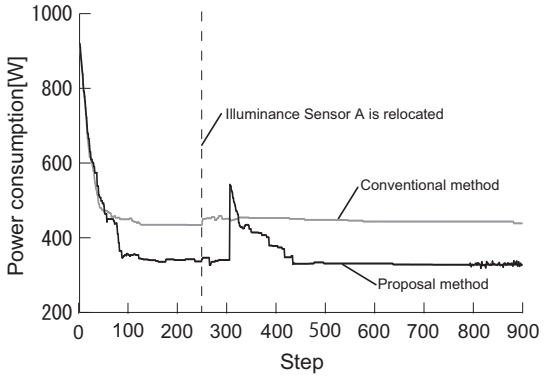


Fig. 11. Historical power consumption by two methods

to measure the levels of influence of those lightings on illuminance sensors.

To solve this problem, we have proposed in this study a new lighting control algorithm which can cope with illuminance sensor relocations by estimating sensor positions using data from turned-on lightings. To verify the effectiveness of the proposed method, an operational experiment was conducted using 15 lighting fixtures and assuming 3 users simulating a real office environment. The experiment demonstrated that the proposed system realizes a performance equivalent to the conventional intelligent lighting system while reducing power consumption.

REFERENCES

[1] Y.Kaneko and S.Kitagami, "Energy-Saving Lighting Control System Linked to Access Control", MITSUBISHI ELECTRIC ADVANCE, Vol. 131,pp.8-11, 2010, http://www.mitsubishielectric.com/company/rd/advance/pdf/vol131/vol131_tr4.pdf.

[2] P. Boyce, N. Eklund and N. Simpson, "individual Lighting Control Task Performance, Mood, and Illuminance", Proc J. of the Illuminating Engineering Society, pp.131-142, 2000.

[3] K.Yamakawa,K.Watabe,M.Inamura and H.Takeda, "A Study on the Practical Use of a Task and Ambient Lighting System in an Office", Proc Journal of light and visual environment vol.24, pp.15-18, 2000.

[4] M. Miki, T. Hiroyasu, and K.Imazato, "Proposal for an Intelligent Lighting System, and Verification of Control Method Effectiveness," Proc CIS, vol.1 pp.520-525, 2004.

[5] Y.Kasahara, M.Miki, M.Yoshimi, "Preliminary Evaluation of the Intelligent Lighting System with Distributed Control Modules",Proc. ISDA2011, pp.283-288, 2011

[6] M.Miki, Y.Kasahara, T.Hiroyasu, M.Yoshimi, "Construction of Illuminance Distribution Measurement System and Evaluation of Illuminance Convergence in Intelligent Lighting System", Proc IEEE Sensors, pp.2431-2434,2010.

[7] S.Inoue, MITSUBISHI ESATE COMPANY Ltd, "Towards the City of the Future" http://www.jetro.org/documents/green_innov/Shigeru_Inoue_Presentation.pdf

[8] S. Tanaka, M. Miki, T.Hiroyasu and M.Yoshikata, "An evolutional optimization algorithm to provide individual illuminance in workplaces",Proc IEEE SMC, pp.941-947, 2009.

[9] Ralph D Ellis, "Illumination guidelines for nighttime highway work", 1995

Real Time Spatiotemporal Biological Stress Level Checking

Marina Uchimura[1], Yuki Eguchi[2], Minami Kawasaki[2], Naoko Yoshii[1], Tomohiro Umeda[3], Masami Takata[4], and Kazuki Joe[4]
[1]Graduate School of Humanities and Sciences, Nara Women's University, Nara, Nara, Japan
[2]Department of Information and Computer Sciences, Nara Women's University, Nara, Nara, Japan
[3]Social Cooperation Center, Nara Women's University, Nara, Nara, Japan
[4]Academic Group of Information and Computer Sciences, Nara Women's University, Nara, Nara, Japan

Abstract— *We are going to investigate and design a healthcare navigation system that consists of sensor devices for human's vital sign, a mobile terminal for transferring the vital data to the cloud, and a cloud computing environment for intelligential processing of the vital data. In this paper, we present a tool for the mobile terminal, i.e. a smartphone, which displays user's biological stress levels. Although the tool is a part of our healthcare navigation system, it does not require the computation resource of the cloud because the stress levels, LF/HF, are enough computable on the processor of the current smartphone in real time. LF/HF is an index of sympathetic nervous activity calculated with the heart rate data obtained from heartbeat sensors. The LF/HF values are displayed in a form of bar graph at user's current location on the Google map of the smartphone. The location information is easily obtained from the GPS of the smartphone. Using the tool, the user can see his/her current stress levels as bar graph on the Google map in real-time. Namely, the tool provides users with their spatiotemporal biological stress levels in real time.*

Keywords: RR interval, WHS-1, smartphone, googlemap, LF/HF

1. Introduction

Over the past 20 years, the CPU performance, the HDD capacity, and the internet bandwidth have improved 1,000 times, 20,000 times, and 15,000 time, respectively. As a result, cloud computing is available among the world. With the rapid expansion of the computation resource, the idea of life computing is aborning. Unlike existing scientific computing, life computing supports our human life. The demand is exceeding the demand of scientific and engineering calculations. We are just going to investigate and design a healthcare navigation system. We believe that this healthcare system will be a major field of life computing. Healthcare computing consists of sensor devices for human's vital sign, a mobile terminal for transferring the vital data to the cloud, and a cloud computing environment for intelligential processing of the vital data. Then this system sends appropriate and useful recommendation about healthcare to the user. In this healthcare navigation system, a mobile terminal gets the vital data from sensor devices to transfer them with applying appropriate pre-processes to the cloud. It is sometimes possible to directly provide the user with information about his/her health condition at the mobile terminal without using computation resource of the cloud. In this paper, we develop a tool for a mobile terminal, i.e. a smartphone, which displays LF/HF that is a method to measure stress levels in real time as a part of our healthcare navigation system. The LF/HF values are displayed in a form of bar graph at user's current location on the Google map of the smartphone. Frequency analysis is applied to time series data for heart rate variability. Its power spectral includes LF that is low frequency and HF that is high frequency. The ratio of LF to HF is used for an index of stress levels (sympathetic nervous activity). Extreme stress causes various physical harms. It is helpful for us to reduce stress for spending healthy life. The tool does not require any computation resource of the cloud because LF/HF are enough computable on the current smartphone in real time. The location information is easily obtained from the GPS of the smartphone.

Instant Heart Rate produced by Azumio[1] and Period tracker[2] produced by GP Apps are examples of existing healthcare services. The former can measure heart rate by using camera equipped with most smartphones. The application measures pulse waves by pressing the camera with a finger. When user inputs the start date of menstruation, the latter can predict the next menstruation and next ovulation date. In these services, user consciously provides his/her vital data for these services. These services influence user's daily life owing to the device size. A healthcare system needs to measure at all times for predicting user's condition accurately. However, the size of the heart rate sensor which is used in this paper is 40.8*37.0*8.9 cubic millimeter and weight is only 13 grams including a battery cell. This sensor can be put on user's body directly at all times. User only has to put on the heart rate sensor to send vital data. This sensor does not influence user's daily life. The tool which we present here notifies user of his/her stress levels without sending user's vital data to the cloud.

In the case that just time and LF/HF are displayed, it is difficult to predict what situations cause stress. Not only time and LF/HF but also other information is required to solve

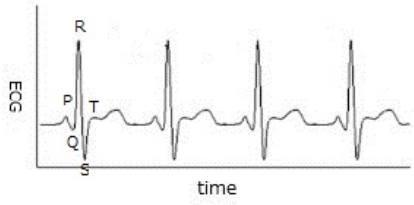


Fig. 1: Heart rate data

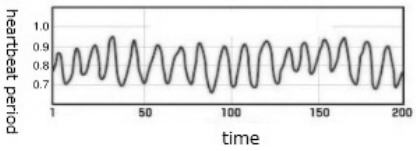


Fig. 2: Time series data of heart rate variability

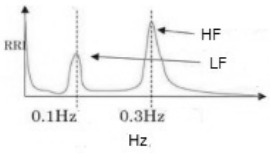


Fig. 3: Frequency analysis

this problem. One of the factors that give stress is location information. Getting stressed or relaxed may depends on the surrounding. The action at the notable time can be predicted from the location information. In this tool, the association of the location information and LF/HF can be predicted in what situations cause stress. It helps to reduce stress for spending healthy life.

The rest of the paper organized as follows. In section 2, LF/HF which is an index of sympathetic nervous activity is described. In section 3, we explain the requirement specification and the implementation of the tool. In section 4, we show some examples of the execution of the tool.

2. Heart rate data

This section explains the heart rate sensor used to take the heart beat and the calculation method for LF/HF as the index of the stress used in this paper.

2.1 Calculation of LF/HF

As shown in Fig.1, various shapes of waves such as P, Q, R, S, and T waves are included in each heart rate. The RR interval is an interval of R waves that is the biggest peak wave in each heart rate. It is known that the RR interval is synonymous with the heart rate, and always changes constantly.

Fig. 4: heartbeat sensor

As for the evaluation of the autonomic nervous function, the index of the frequency domain obtained by the spectral analysis is used [3][4][5]. Figure 2 shows time series data of heart rate variability. The horizontal axis and the vertical axis are time and RR interval, respectively. Figure 3 shows frequency analysis of time series data of heart rate variability. Its power spectral includes LF which is low frequency and HF which is high frequency. LF is related to the change of blood pressure and influenced by both parasympathetic and sympathetic nervous system. HF is related to breathing variation and influenced by parasympathetic nervous system. The sympathetic nervous system promotes the shrinking of the blood vessel, the increase of the heart rate, and the reaction of the muscular tension etc., so that it reacts to the change in the situation quickly, and the stress like uneasiness, fear, and anger, etc. has been received. Parasympathetic nervous system promotes responses which include the decrease of the heart rate, decreased blood pressure, and muscle relaxation, etc. Sleeping and getting relaxed, parasympathetic nervous system works.

The index of the sympathetic nervous system is calculated with the ratio of LF to HF. HF decreases while LF appears when a stress is received. When parasympathetic nervous system becomes active, HF increase. Therefore, getting stress and relaxed make LF/HF large and small, respectively.

LF/HF is calculated with time series data of heart rate variability. The time series data of heart rate variability is applied to linear interpolation or spline interpolation to get equal interval samples. Power spectrum of sampling data is calculated using Fourier transform or an autoregressive model. LF and HF are set to area of 0.05Hz - 0.15Hz and 0.15Hz - 0.40Hz, respectively.

2.2 Heart rate sensor

Figure 4 shows WHS-1 manufactured by union tool corporation [6] that is the heart rate sensor we use. This sensor detects and calculates the RR interval. It is possible to use it without interfering in user's daily life by putting directly on the skin with an electrode pad because the size is so small (40.8*37.0*8.9mm), and the weight is very light (only 13 grams) including a battery cell. WHS-1 also includes an

acceleration sensor and a temperature sensor as well as the heart rate sensor so that it detects the change of user's body movement and measures his/her body surface temperature.

WHS-1 supports two types of data transmissions. One method is sending data to a PC using wireless communication. The other method is sending data to a PC using a flash memory. In the case of wireless communication, it can also output the heart rate waveform instead of the RR interval.

In the case of outputting RR interval, heart rate signal, which are measured by an electrode attached to the sensor, is sampled at 1,000Hz. Then RR interval is calculated by predicting the interval between R waves which are found in the heart rate waveform. A flash memory with16MB saves almost one week RR intervals data.

3. Displaying LF/HF in real time

In our future healthcare navigation system, vital data is transferred to the cloud via a smartphone. A smartphone can effectively utilize the information from the sensor device. In this paper, LF/HF is calculated with the vital data, and GPS equipped with the smartphone is used. We develop a tool that displays LF/HF values on the map at the user's location in real time. In this section, we explain the requirement specification and the implementation issues of the tool.

3.1 Requirement specification

The stress might be received in unconsciousness to cause deconditioning as a result while we learn the condition of our daily life by knowing the changing of LF/HF values. However, this is just an understanding when to have caused the stress and the relaxation. Since it is difficult to know which situation has caused the stress, not only time but also other information are necessary. By using time and other information, we can predict the factors which bring down the stress. As such extra information, location information is easily given with GPS, for example. The LF/HF value and the location information help us predict our condition at that time. Then the relation between our stress and our daily life is clarified. In the proposed tool, the LF/HF value should be displayed on the map at user's current location.

Figure 5 shows the correlation of each function. LF/HF value is calculated and displayed by numerical characters every 10 seconds. Moreover, the value is shown with a bar graph at user's present location on the Google map of a smartphone every 60 seconds. When user swipes on the screen, the map works with the swipe. When user touches on the screen, a zoom item is displayed and the map can be zoomed in or out. When the menu button is pushed, the MyLocation item, the Reset item, and the On/Off item are displayed. When the MyLocation item is selected, the present location is displayed at the center of the screen. When the Reset item is selected, the LF/HF bar graph displayed on the screen is deleted. When the On/Off item is selected, the bar graph is displayed or hidden on the screen. The specification to achieve the above-mentioned functions is as follows.

1) The LF/HF value is displayed in digits every 10 seconds.
2) The LF/HF bar graph is put at the present location on the Google map every 60 seconds.
3) User's movement information on speed, distance, altitude, and traveling direction, etc. are displayed.
4) Swipe of the map
5) Zoom of the map
6) In the case of the expanded map, it switches from the bar graph to the label.
7) The present location is obtained.
8) The LF/HF value is display or hidden.
9) The displayed LF/HF value can be deleted.
10) The mobile terminal and the sensor are communicated with Bluetooth.

In function 1), the LF/HF value that presents stress level is displayed and updated every 10 seconds so that user can confirm the LF/HF every 10 seconds.

In function 2), user can confirm the change of his/her stress level by looking at the displayed LF/HF bar graph every 60 seconds. The more heart rate data, the more accurate LF/HF is calculated. When the LF/HF value is displayed as numerical characters, it is difficult to confirm the change of the LF/HF value viscerally. To solve this problem, a form of bar graph is adopted. A series of bars along user's move makes the user confirm the change of the LF/HF value viscerally. User can understand what situation causes a stress by confirming the change of the LF/HF value which is drawn at user's current location on the Google map at that time.

It is known that exercise influences LF/HF. So the relation between exercise and LF/HF is effectively predictable by giving the information of user's exercise. In function 3), exercise information such as speed, distance, elevation and direction are displayed.

In function 4), when user swipes the screen, the Google map moves accordingly. The user can see the LF/HF value of any location by using swipe.

In function 5), the Google map can be zoomed in or out. When the wide range is displayed, the change of the LF/HF value can be roughly confirmed. On the other hand, when a small range is displayed, the relation between the place and the LF/HF value can be observed in detail. When two or more bar charts are displayed by the large scale, the LF/HF value is not confirmed easily. Then, in function 6), it switches to the label from the bar chart to the integer portion of the LF/HF value.

In function 7), user's current location is put at the center of the screen using the GPS of the smartphone so that he/she confirms his/her location any time.

In function 8), user can display or delete the LF/HF bar graph/numeric characters. When he/she wants to see just the

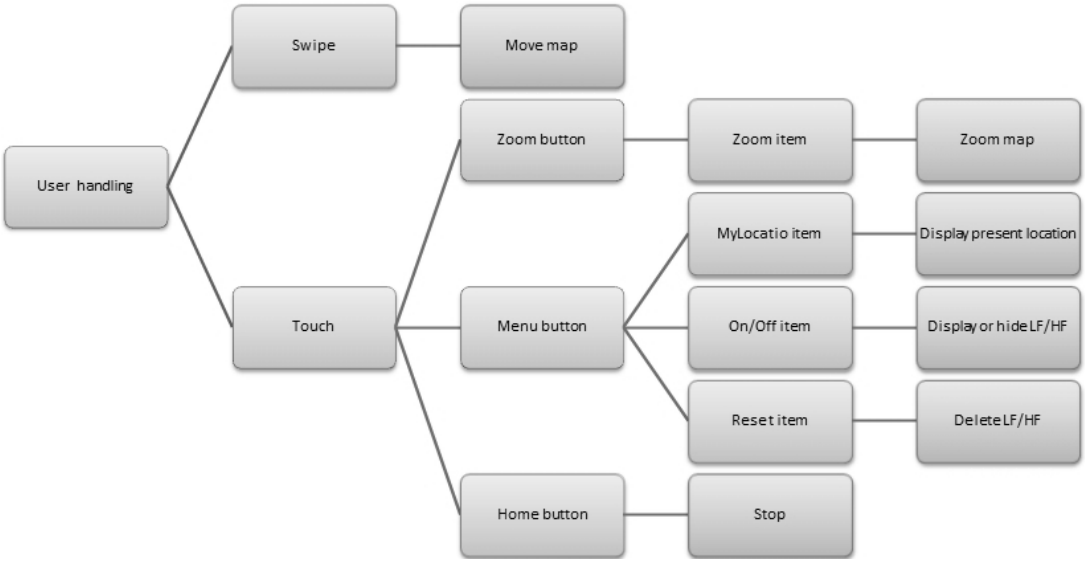


Fig. 5: correlation of each functions

Google map, the display of the LF/HF value may make the screen hard to see.

Function 9) is used for deleting the record of the LF/HF value data. When a lot of the LF/HF value data is recorded, various functions might not be able to be operated smoothly. Function 9) is used to prevent such cases.

The tool we present in this paper is executed on a mobile terminal emulator on a PC, and does not use the Bluetooth communication. Since this is a hardware restriction, we will easily support the Bluetooth communication very soon using a real mobile terminal.

3.2 Implementation

When our tool is invoked, user's current location is displayed by using GPS on the Google map. Then, it starts reading the heart rate data from WHS-1. The LF/HF value is displayed and updated by digit in the upper part of the screen every 10 seconds. In addition, the current location is displayed at the center of the screen, and user's moving information of speed, distance, altitude, and direction are displayed, too. The LF/HF bar graph is put on the Google map to be updated with an interval of 60 seconds. The acquisition of the current location, the calculation of LF/HF, and the explanation of GUI are presented as follows.

3.2.1 Acquisition of user's current location

The GPS of the smartphone is used to acquire user's (namely smartpone's) current location. The application makes use of Google Map API [7]. Google Map API provides Maps API and Maps Javascript API that are widely used for smartphones. In this paper, we adopt Maps API because Maps API Key is necessary to control the Google Map. When the location information is required, an API called Location-based Service that enhances GPS functions is used in smartphones. The GPS acquires the location information by way of LocationManager. The getSystemService method obtains LocationManager, and begins to acquire the location information of latitude, longitude, altitude, direction, and speed, etc. The update notification for the location information is set by the requestLocationUpdate method, and sent to the LocationListener interface. Function 7) is implemented with these APIs.

For the requirement specification regarding to the location information (Function 3)), an object of the Ctiteria class is sent to the Provider as a parameter. The distanceBetween method calculates the distance from the starting point to the current point.

3.2.2 LF/HF calculation implementation

In this subsection, we explain the implementation issues for calculating LF/HF. X and Y are provided for time and the RR interval, respectively. Then, a spline interpolation is applied to the sampling data with an equal interval. The power spectrum is calculated from the sampling data with the equal interval by a Fourier transform. The areas of LF and HF are 0.05Hz - 0.15Hz and 0.15Hz - 0.40Hz, respectively.

Heart rate data is sent from WHS-1 every 3 heart beats. Then the received data is stored in a circular buffer, which is implemented with an array and modulo operations for the array index. Using the circular buffer, the data which become unnecessary is overwritten with new data to avoid consumption of memory. As is easily expected, the more heart rate data, the more accurate LF/HF is calculated. We investigate the tradeoff between the circular buffer size and

the LF/HF accuracy to decide the buffer size for 60 seconds data. The initial LF/HF is calculated with the data of first 60 seconds. After the first calculation, LF/HF is calculated every 10 seconds using the last 60 seconds data, which occupies the circular buffer. Each new 10 seconds data is overwritten to the oldest 10 seconds data.

In this way, LF/HF is obtained every 10 seconds, so function 1 is satisfied. In addition, LF/HF is also obtained every 60 seconds, so Function 2) is satisfied.

3.2.3 GUI

In the screen of the smartphone, the information of LF/HF, speed, distance, elevation and direction is displayed on the upper part. Since the information is updated every 10 seconds, Function 3) is satisfied. The Google map is displayed below the information area. The MyOverlay class is used to draw a bar graph of LF/HF at the current location on the Google map, and Function 2) is satisfied.

The Google map moves at the same time as user swipes. When user swipes, the touching point moves with user's finger. User can move the Google map to confirm the LF/HF value anywhere. Namely, Function 4) is achieved.

The SetBuiltInZoomControls method is used for the expansion and reduction of the Google map to satisfy Functions 5) and 6). The LF/HF value is expressed as digit label or bar graph depending on the scale of the Google map. The digit label is the current LF/HF value while the length of the bar graph is the LF/HF value at that point. In the case LF/HF is expressed as bar graph on the small scale of the Google map, the LF/HF value is not confirmed easily. Moreover, in the digit label on the large scale map, the change of the LF/HF values is not confirmed easily. The onZoomListener method is needed to solve this problem.

The onCreateOptionsMenue method is used for setting menu buttons equipped with smartphones. When the menu button is pushed, menu items which include MyLocation, Reset and On/Off are displayed. When user select an item, the Item.getItemId method judges which item is selected. The selected item is processed by the onOptionsItemSelected method. When the Mylocation item is selected, the current location is obtained to be displayed in the center of the screen. When the Reset item is selected, the record of the LF/HF values is deleted. When the On/Off item is selected, the LF/HF value is displayed or hidden. In this way, Functions 7)-9) are satisfied.

4. Example of behavior

In this section, we explain the performance of our tool that is an Android application. The tool is developed with Java SE Development Kit, Android SDK, and Eclipse. Java SE Development Kit is used for developing Java applications. Android SDK is for the development of Android applications by Java. Eclipce is an integrated development environment. The heart rate data is obtained from WHS-1.

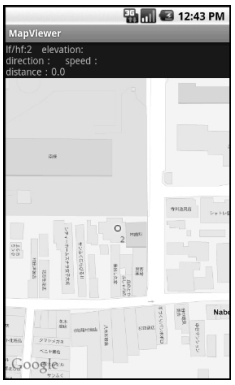


Fig. 6: The screen of the tool at 60 seconds after the invocation

(a)initial state (b)result

Fig. 7: swipe

In this paper, we use heart rate data which is preliminarily obtained from heart rate sensor and display the LF/HF bar graph at arbitrary location. We believe it is enough to investigate whether Function 1)-9) are achieved.

Invoking the tool, the screen after 60 seconds is shown in Fig. 6. Each LF/HF is calculated and updated every 60 seconds. Since the LF/HF bar graph is put at current location, user can predict the correlation between stress and location information. The LF/HF value shown in Fig. 6 is calculated with the data of the initial 0-60 seconds. In the upper part of the screen, LF/HF, altitude, speed, direction, and moving distance are displayed and updated every 10 seconds.

Figure 7 shows the change of the screen by swipe. Figure 7(a) is the initial state. When the screen is swiped from right to left, the screen moves left according to the swipe. Figure 7(b) shows the result of the swipe. Using swipe, user can confirm his/her LF/HF vale at any location.

In Fig. 8, when user touches the screen, the zoom item is displayed. Figure 8(a) is the initial state. When the plus icon in the zoom item is selected, the Google map is zoomed in. Figure 8(b) is the zoomed screen by the plus icon. On the other hand, when the minus icon in the zoom item is

(a)initial state

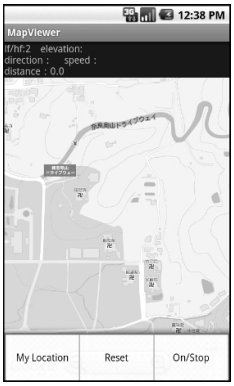
(b)result

Fig. 8: zoom button

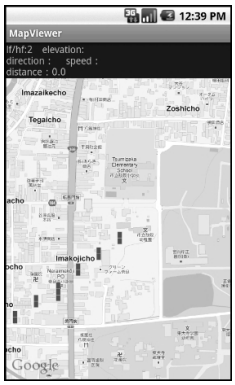
(a)initial state

(b)result

Fig. 10: On/Off button



(a)initial state



(b)result

Fig. 9: MyLocation item

(a)initial state

(b)reesult

Fig. 11: Reset button

selected, the Google map is zoomed out. The LF/HF values are shown as bar graphs on figure 8(a) while the LF/HF values are shown as the digit label as on figure 8(b). The toggle for the bar graph and the digit label is the zoom scale of the Google map. The LF/HF bar graph changes to the LF/HF digit label when the zoom scale exceeds the pre-defined value. The length of the bar graph presents the LF/HF value at that point. On the small Google map, two or more bar graphs represent the change of the LF/HF values viscerally. User can understand where his/her stress arises. On the other hand, the digit label presents the integer part of the LF/HF value at that time. On the large Google map, user's past and present locations are displayed in detail. The user understands the relation between the LF/HF value and his/her moving path. Therefore, the switch of the bar graph and the digit label is useful to understand what situation causes his/her stress.

Figure 9 shows the screen applied by the MyLocation item. When the menu button of the smartphone is pushed, the menu item is displayed at the bottom of the screen as shown in Fig.9(a). In the menu item, the MyLocation item, the On/Off item, and the Reset item are included. When the MyLocation item is selected, the current location is acquired to be displayed at the center of the screen. In Fig.9(a), the current location is not displayed. When the MyLocation item is chosen, the current location is displayed at the center of the screen as shown in Fig.9(b). User can confirm his/her current location at any time.

Figure 10 shows the screens where the On/Off item is used. When the On/Off item in the menu item is selected, the LF/HF bar graphs are displayed or hidden. Fig.10(a) shows the screen with the LF/HF bar graphs. Using the On/Off item, the LF/HF bar graphs are hidden as shown in Fig.10(b), and user can take a detail look at the Google map without the LF/HF bar graphs. When the LF/HF bar graphs are hidden, the LF/HF value is calculated every 10 seconds. Using the On/Off item, the LF/HF bar graphs appear again.

Figure 11 shows the screens where the Reset item used. When the Reset item is selected, the LF/HF bar graphs as shown in Fig.11(a) are deleted. Then, the LF/HF bar graphs begin to be displayed. When a lot of LF/HF bar graphs are displayed, various functions might not be able to be operated smoothly. This item is used to avoid this problem.

5. Conclusions

In our ongoing healthcare navigation system, vital data from sensor devices are sent to a portable terminal such as smartphone to process the data including error correction. The processed vital data are sent to a cloud for intelligent processes. In this paper, we showed that it is possible to show some of user's conditions to the user without cloud resource, but with just a smartphone.

In this paper, we presented a tool for a mobile terminal that displays the LF/HF value in real time as an index of user's stress level. The LF/HF value is calculated by using heart rate data obtained from a heart rate sensor WHS-1, and displayed at user's current location in the Google map. Some physical conditions cause various slumps by an excessive stress. User can understand what situation causes his/her stress by using the location information and the LF/HF values. When the source of stress is detectable, user can easily improve his/her daily life. It leads to the environment improvement to find the reason of stress brought on by location such as noise.

In this tool, the LF/HF value is immediately calculated every 10 seconds. To confirm the change, the LF/HF bar graph is drawn on the Google map. The information of speed, distance, elevation and direction is displayed to predict the relation between the exercise and the LF/HF value effectively. The Google map can be zoomed in/out, and swiped freely. The LF/HF value is shown in a form of bar graph to understand the relation between user's location and LF/HF value in the small scale Google map, and the LF/HF value is shown in the digit label in the large scale. The user can confirm his/her current location at any time. When user wants to look at the Google map, the LF/HF value can be hidden. To operate various functions smoothly, the LF/HF value record can be deleted. In this paper, we develop a prototype to show the actual execution.

For our future work, it is necessary to include other information such as user's acceleration and some environment information such as weather. The information helps user predict his/her situation more precisely. Another method to calculate the stress level with electro-oculogram is needed to improve the accuracy of the stress level.

References

[1] (2012) azumio website. [Online]. Available: http://www.azumio.com
[2] (2012) Apple website. [Online]. Available: http://itunes.apple.com/us/app/period-tracker-deluxe/id289084315?mt=8
[3] Pomeranz, B., Macaulay, R.J., Caudill, M.A., Kutz, I., Adam, D., Gordon, D., Kilborn, K.M., Barger, A.C., Shannon, D.C., Cohen, R.J., and al. et.: Assessment of autonomic function in human by rate spectral analysis, American Journal of Phtsiology, vol. 17, pp. 151-153 (1985).
[4] S-H Chang, C-H Luo, and T-L Yeh: An experimental design for quantification of cardiovascular responses to music stimuli in humans, Journal of Medical Engineering & Technology, Vol. 28, No. 4, pp.157-166 (2004).
[5] Baharav, A., Shinar, Z., Akselrod, S., Mosek, A., and Davrath, L.R.: Cluster headache patients have normal circadian and sleep time autonomic nervous system function, Inproceedings of Computers in Cardiology 2005, pp.263-266 (2005).
[6] (2012) UNION TOOL CO. website. [Online]. Available: http://www.uniontool.co.jp/english/index.html
[7] (2012) Google Maps API Family website. [Online]. Available: http://code.google.com/intl/en/apis/maps

AR based Spatial Reasoning Capacity Training for Students

Mai Hatano[1], **Tomoko Yonezawa**[2], **Naoko Yoshii**[1], **Masami Takata**[3], **and Kazuki Joe**[3]
[1]Graduate School of Humanities and Sciences, Nara Women's University, Nara, Nara, Japan
[2]Faculty of Informatics, Kansai University, Takatsuki, Osaka, Japan
[3]Academic Group of Information and Computer Sciences, Nara Women's University, Nara, Nara, Japan

Abstract— *In this paper, we propose two methods to train students' spatial reasoning capacity using AR (Augmented Reality). The first method supports students for rotating spatial objects more easily with two AR markers. One marker is used for questions, on which several blocks and a landmark (with a shape of a chick) are displayed. The other marker is used for answers, on which blocks are moved freely. The layout of the blocks toward the chick is selected on the marker. The second method includes limitation of rotation on the marker using some Arduino based hardware. The second method supports students for rotating spatial objects partially. To validate the effect of the trained and resultant spatial reasoning capacity, we perform an experiment using the first method. The analysis results explain the spatial object recognition accuracy increases using the AR learning. To validate the effect of rotation angles, we perform other experiments using the second method. The analysis result shows the rotation angle of sixty degrees is the best for the training of spatial reasoning capacity.*

Keywords: Augmented Reality, Spatial reasoning capacity, Mental rotation, Arduino

1. Introduction

The spatial reasoning capacity is an ability to recognize objects in three-dimensional space. Objects' locations and conditions, which are shapes, angles and sizes, are recognized quickly and accurately by this capacity. Even non-existent objects are always imaged using the spatial reasoning capacity, too. Visual images can be controlled by cognitive operations in the same way for real objects.

Compulsory education includes the learning of spatial objects in some ages although the learning of the spatial objects has been decreased according to lighter curriculum promoted by the Ministry of Education in Japan. In the second grade of elementary school, children learn spatial objects for the first time as compulsory education [1]. First, children learn the basic objects, which are cubes and rectangular solids. These objects are shown in learning materials such as textbooks with a plane surface. Consequently, if they do not have spatial reasoning capacity so much, it is difficult to learn abecedarian arithmetic. In such case, they cannot understand spatial objects in junior high school. Moreover, selected specific general planes and equations for lines are given in mathematics of high school regardless of their intelligibility of spatial objects.

Children in old days mainly used to play with three-dimensional objects, such as building blocks, cat's cradle, puzzle links and plamodels. On the other hand, children in nowadays mainly plays in a plane, i.e. video games, rather than in the three-dimensional world. Spatial reasoning capacity is improved by strong awareness in the three-dimension space, for example, playing with three-dimensional objects. Opportunities of spatial reasoning learning have decreased in terms of play and education in general. Hence, special education for spatial reasoning capacity is needed in childhood.

In this paper, we propose some methods to train students' spatial reasoning capacity with using AR. In the AR environment, virtual objects are displayed even in floating and can be moved freely. Students with low spatial reasoning capacity can effectively learn by rotating an AR marker. To validate the effects of the trained and resultant spatial reasoning capacity, we perform some experiments and analyze them to model the learning phases.

The rest of the paper is organized as follows. In section 2, we propose two methods to train students' spatial reasoning capacity using AR. In section 3, the effect of the trained and resultant spatial reasoning capacity is validated. In section 4, the effect of rotation angle to the improved spatial reasoning capacity is validated to be represented by a model expression.

2. Rotating blocks using AR

In this section, two methods to train spatial reasoning capacity are proposed. These methods support easy spatial object rotation for improving spatial reasoning capacity. We first explain the construction of an AR supporting environment. Then, we propose the first method to train spatial reasoning capacity and the second method with physical limitation of rotation.

2.1 Construction of an AR supporting environment

In this paper, we adopt AR to provide free spatial objects movement with displaying the corresponding virtual objects for students. In the AR space, the acquisition of virtual object location is generally performed by the following two ways.

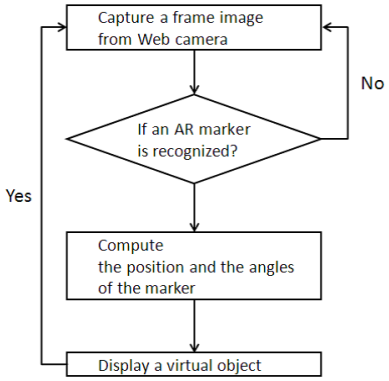


Fig. 1: Construction of the AR environment

One is by using GPS (Global Positioning System). The other way uses an AR marker on which an asymmetry figure is printed. AR markers are freely movable, and virtual objects on the AR markers are redisplayed according to their movement. Since we are interested in training spatial reasoning capacity, we use the marker based AR that provides the free movement of virtual objects for students.

The marker based AR supporting environment for spatial reasoning capacity is constructed as shown in Fig.1. First, a frame image of video stream from a Web camera is captured to be displayed. Next, an AR marker is recognized in the captured image. Then, the position and the angles of the marker are computed. Finally, a pre-defined virtual object is displayed at the position with the angles. The AR marker receives events from user via keyboard to change the position and the angles of the virtual object.

2.2 Rotating virtual objects in the marker based AR supporting environment

In this subsection, we present a method to train students' spatial reasoning capacity with rotating virtual blocks in the above AR environment. This method requires two AR markers: for questions and for answers. On the question marker, several blocks, a landmark (with a shape of a chick) and box frames are displayed. Figure 2 shows an example where two blocks and a landmark are displayed as a question. Up to 2^3 blocks can be placed in the box frame. The number of blocks is selected using the keyboard. The layout of blocks is changeable in random. The landmark can be placed on the four directions. The box frame is always displayed. The possible number of questions is $2^8 * 4 = 1,024$.

On the answer marker, numbers of 1 to 4 are written. First, only the box frame is visible on the answer marker. A block is placed on the answer marker according to the keyboard input. The keyboard input is chosen from key 1 to 4. When a key is pressed, a block is placed on the same number of the answer marker. Figure 3 shows examples of the key entries. When key 1 is pressed once, a block is displayed on

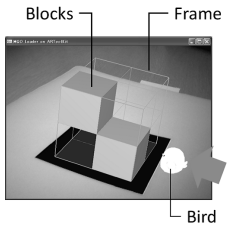
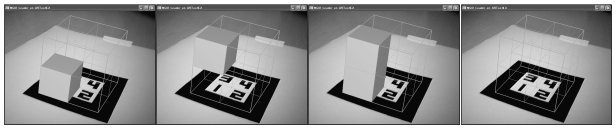


Fig. 2: A question marker



(a) (b) (c) (d)

Fig. 3: Block placement ((a):One Click, (b):Two Clicks, (c):Three Clicks, (d):Four Clicks)

(a) Correct (b) Incorrect

Fig. 4: Answer markers

the bottom as shown in Fig.3(a). When the key is pressed twice, the block moves up as shown in Fig.3(b). When the key is pressed three times, two blocks are displayed on the bottom and the top as shown in Fig.3(c). When the key is pressed four times, the two blocks disappears and the key entries are canceled as shown in Fig.3(d).

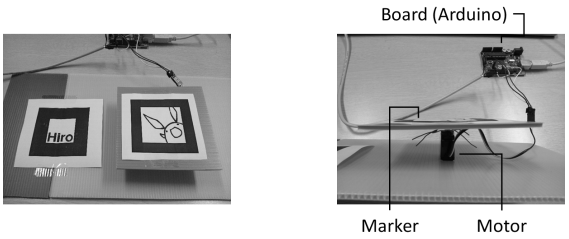
Figure 4 shows two examples of answers. The layout of the blocks toward the landmark is selected on the answer marker. The correct and the incorrect layouts of blocks are Fig.4(a) and (b), respectively. When the correct layout is given, the color of the displayed blocks is changed.

These blocks move synchronously with the marker. This method is applied to varous rotations of blocks. When a given question is hard to be solved, the marker can be rotated physically with the displayed blocks so that the question becomes easier.

2.3 Improved version of the AR environment

The method described in subsection 2.2 does not take the rotation angle of the AR marker in account, and blocks can be observed from any direction. In this subsection, we improve the first method so that it provides analysis functions for the relationship between the angle and the learning efficiency of students.

The improvement to the AR environment is given by



(a) Top view (b) Side view

Fig. 5: Views of AR markers installed with the moter

restricting the rotation of the AR marker supported by some hardware module. The hardware module consists of a fixed board (a plastic cardboard), a servo motor RB-001 and an Arduino Uno [3] based substrate. Arduino Uno is used as an A/D converter, which controls the servo motor, and is connected to the host PC via USB interface. Figure 5 shows an answer marker and a question marker installed with the motor. Figure 5(a) and 5(b) show the top and the side view, respectively.

We show the procedure to control the question marker.

Step-1) Power down the motor
Step-2) Compute the angle of the question marker
Step-3) When the angle of the marker does not reach to a given threshold, go back to Step-2)
Step-4) Start serial communication
Step-5) Power up the motor
Step-6) Push back the marker
Step-7) Power down the motor and go back to Step-2)

In Step-1), the question marker can be rotated freely since the motor is off. In Step-2), to compute the angle of the marker, we get the marker coordinate (X_m, Y_m, Z_m) and the camera coordinate (X_c, Y_c, Z_c). The two coordinates are expressed as

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_1 & r_2 & r_3 & t_x \\ r_4 & r_5 & r_6 & t_y \\ r_7 & r_8 & r_9 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix}, \quad (1)$$

where t_x, t_y and t_z are translation elements. $[r_1, r_4, r_7]^\top$, $[r_2, r_5, r_8]^\top$ and $[r_3, r_6, r_9]^\top$ mean directional vectors of x-, y- and z-axis, respectively. Eulerian angles are computed with the directional vectors to be used for the marker angle. In Step-3), the marker angle is compared with a given threshold, which restricts the rotation of markers. When the marker angle reaches to the threshold, serial communication between the host PC and Arduino via USB is started in Step-4). At this point, Arduino is notified of the marker angle violation. So it turns on the motor in Step-5), and controls the motor so that the marker angle does not exceed the threshold in Step-6). When the marker angle becomes less than the threshold, Arduino turns off the motor and wait for the next marker angle violation in Step-7).

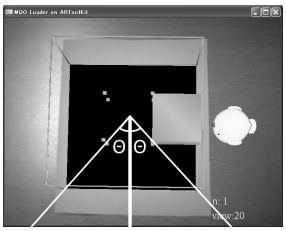


Fig. 6: A view of the angle limitation for a question marker

Figure 6 shows a view of the angle limitation for a question marker. The location of the landmark is set to the right direction of the marker unlike in subsection 2.2. The angle limitation is defined as θ ($0 \leqq \theta \leqq 90$).

3. An experiment to validate the effect of the AR learning

To validate the effect of the trained and resultant spatial reasoning capacity, we perform an experiment using the method described in subsection 2.2. The method is implemented with ARToolKit [4] to provide an AR space. Objects in the AR space are displayed by using OpenGL [5].

According to Piaget's stage of cognitive development [6], four cognitive stages, which are a sensorimotor stage, a pre-operational stage, a concrete operational stage and a formal operational stage, are observed. The sensorimotor stage is the period of zero to two years old, where a hidden object can be recognized and the permanence of an object is acquired. The pre-operational stage is the period of two to seven years old, where limited intelligence is developed to symbolize objects so that make-believe games are played. On the other hand, thinking is done without logic. The concrete operational stage is the period of seven to twelve years old, where considerable intelligence is developed through logical and systematic manipulation of objects. Finally, complete intelligence, which is as competent as adults, is established in the formal operational stage at the age of twelve. The spatial reasoning capacity is developed by training. Since objects, which are generated in the method in section 2.2, are examined and logicized, children in concrete operational stage should be targeted. Therefore, in this experiment, we chose fifth grade pupils of an elementary school. The pupils are in the age of ten to eleven. In the fifth grade, there are 37 boys and 36 girls.

We start the experiment with the hypothesis that the method using AR does not affect the spatial reasoning capacity. The hypothesis is rejected later in this section.

The conditions of the AR learning are classified into three groups. The first group uses the method in Fig.2, namely "*with frame*". The second group uses the question marker "*without frame*". The effect of box frames shown in Fig.2 is validated by comparing these two groups. Finally, the group

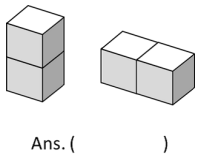


Fig. 7: An example of tests

Fig. 8: A view of the AR learning

of "*without learning*" does not contain the AR learning phase to validate the effect of AR learning.

The experimental procedure is shown as follows.

Step-I) Paper test 1 (5 min.)
Step-II) AR learning (15 min.)
Step-III) Paper test 2 (5 min.)

In Step-I), the preliminary spatial reasoning capacity is measured using test 1. Figure 7 shows an example of paper tests. The right-and-left objects are compared and the pupils determine if these objects are same or different. There are 80 questions in each paper test. The paper test 1 is to be completed within 5 minutes. Note that it is not necessary to answer all the 80 questions. In Step-II), the AR learning is performed in 15 minutes. The pupils of the "*without learning*" group wait 15 minutes without the AR learning. In Step-III), the spatial reasoning capacity is measured using paper test 2 in common with paper test 1. The effect of the AR learning is expected to be validated by this paper test when it is confirmed that Step-I) affects Step-III). These two paper tests are distributed at random.

In this experiment, laptop computers, AR markers and Web cameras are used for the AR learning. The AR markers are freely movable. Figure 8 shows a view of the AR learning.

We analyze the results of the experiment. The spatial reasoning capacity is to recognize the overview of objects in the three-dimensional space quickly and accurately. The spatial object recognition speed is measured with the response rate (the rate of the response time in all questions). In addition, the spatial object recognition accuracy is measured with the accuracy rate. Figure 9 and 10 show the average response rate and the average accuracy rate, respectively. ANOVA (ANalysis Of VAriance) is used to uncover the attributions and interaction effects. Table 1 shows the result of the two-way ANOVA. Here, F shows the proportion of within-group variance to between-group variance. p shows significance probability. * shows significance level. t is calculated with $\frac{\overline{X}-\mu}{\sqrt{U/n}}$, where n is the number of samples, $\overline{X}$ is the sample average, U is the sample unbiased estimate of variance, and μ is the population mean. One attribution is the conditions for learning (learning with frame, learning without frame, and no learning). Another attribution is "*before and after*" (Step-I) and Step-III)).

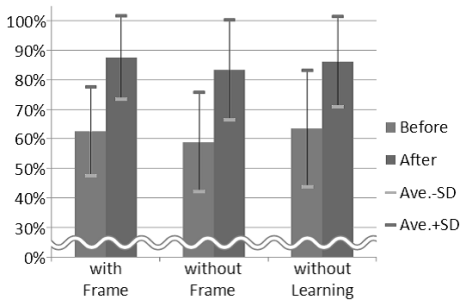


Fig. 9: The response rate

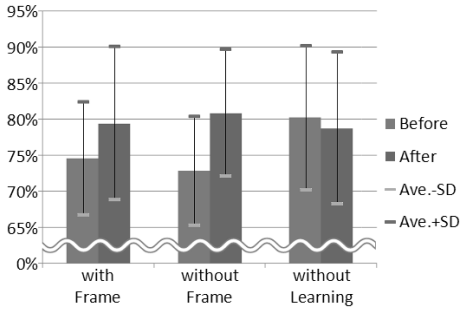


Fig. 10: The accuracy rate

Table 1: Two-way ANOVA

		A response rate	An accuracy rate
Condition	$F(2,70)$	0.54	0.76
	p	0.59	0.47
before and after	$F(1,70)$	189.37	14.82
	p	$< .001$****	$< .001$****
Interaction	$F(2,70)$	0.17	8.09
	p	0.84	$< .001$****

Significant level $p < .001$****

In the result of the response rate, "*before and after*" learning differs significantly ($F(1,70) = 189.37$, $p < .001$****). Therefore, the spatial object recognition speed has the possibility to be improved by the tests. In the result of the accuracy rate, "*before and after*" learning differs significantly ($F(1,70) = 14.82$, $p < .001$****). Interaction between "*before and after*" learning and the learning conditions also differ significantly ($F(2,70) = 8.09$, $p <$

Table 2: The result of a post-hoc test for "*before and after*" learning

An accuracy rate	$F(1,70)$	p
with frame	8.14	$< .01$**
without frame	22.08	$< .001$****
without learning	0.78	0.38

Significant level $p < .01$**, $p < .001$****

Table 3: The result of a post-hoc test for learning conditions

An accuracy rate	$F(2,140)$	p
before	4.13	$< .05$*
after	0.32	0.73

Significant level $p < .05$*

Table 4: The result of a post-hoc test in "*before*" learning

An accuracy rate			t	p
without learning	-	*without frame*	2.76	$< .01$**
without learning	-	*with frame*	2.12	$< .05$*
with frame	-	*without frame*	0.64	0.53

Significant level $p < .05$*, $p < .01$**

.001****). Therefore, the interaction should be evaluated. The results of the simple main effect test are shown in Tab.2 and Tab.3. Significant difference is found in "*before and after*" learning within the "*with frame*" ($F(1,70) = 8.135$, $p < .01$**) and the "*without frame*" ($F(1,70) = 22.083$, $p < .001$****). So the AR learning affects the accuracy rate among "*with frame*" and "*without frame*".

The learning of "*before*" differs significantly ($F(2,140) = 4.13$, $p < .05$*) in the learning conditions. The result of multiple comparisons in the learning conditions of the accuracy rate is shown in Tab.4. The relationship between "*without learning*" and "*without frame*" shows significant difference ($t = 2.76$, $p < .01$**), and the relationship between "*without learning*" and "*with frame*" shows significant difference, too ($t = 2.12$, $p < .05$*). Therefore, it turns out that the group of "*without learning*" contains more examinees with high spatial reasoning capacity. Namely, the hypothesis is rejected. Thus, the effectiveness of the attribute "*with frame*" and "*without frame*" is validated.

As the result of the ANOVA, it is confirmed that the response rate is improved. However, the response rate is improved with any conditions and does not differ significantly. It means that the improvement is not affected so much by the AR learning but the habituation to the tests. Therefore, the spatial object recognition speed is not affected by the AR learning. The accuracy rate is improved by "*before and after*" AR learning. In the case of the accuracy rate, both of "*with frame*" and "*without frame*" are improved while the effect of the box frame is not observed. Thus, the AR learning affects the spatial object recognition accuracy.

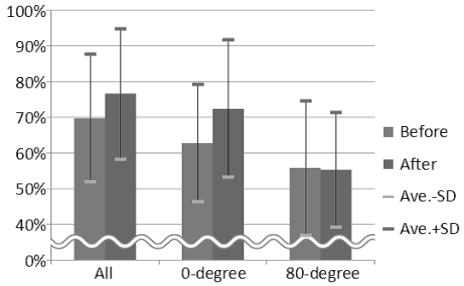


Fig. 11: The response rate in the improved version

4. Experiments for AR marker learning with angle rotation

To study the relationship between rotation angle and the learning efficiency of examinees, we perform other experiments as described in subsection 4.1. In this section, we first investigate the difference between the cases of fixed angle and variable angles for the AR marker by experiment. Then, we explain the experimental results to show the effect of the learning with rotating the AR marker.

4.1 Learning with limited angles

In this subsection, we perform an experiment to investigate the difference between the cases of fixed angle and variable angles. For the experiment, we have 42 examinees who are female students of our university. We start the experiment with the hypothesis that rotation angle does not affect the spatial reasoning capacity using the method of section 2.3. The hypothesis is rejected later in this section.

The experimental procedure is the same as in section 3. Therefore, just different parts are shown as follows. In step II), the method of section 2.3 is used. The conditions of the AR learning are separated into three groups to investigate the angle rotation effect. The three groups are as follows.

- "*all*" (zero, twenty, forty, sixty and eighty degrees)
- "*0-degree*"
- "*80-degree*"

The first group "*all*" has limitations of rotation angles (θ in Fig.6) from zero (cannot rotate) to eighty degrees by twenty degrees. The second group "*0-degree*" is fixed θ zero degree. Finally, the group "*80-degree*" is fixed θ eighty degrees.

We analyze the results of the experiment. Figure 11 and 12 show the average response rate and the average accuracy rate, respectively. Table 5 shows the result of the two-way ANOVA. One attribution is the conditions for learning (learning all, learning with 0-degree, and learning with 80-degree). Another attribution is "*before and after*" (Step-I) and Step-III)).

As the result of the response rate, "*before and after*" learning differs significantly ($F(1,39) = 12.60$, $p < .001$****). Interaction between "*before and after*" learning and the

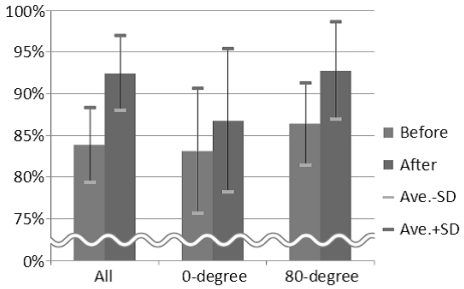


Fig. 12: The accuracy rate in the improved version

Table 5: Two-way ANOVA

		A response rate	An accuracy rate
Condition	$F(2,39)$	3.86	2.57
	p	$< .05$*	0.09
before and after	$F(1,39)$	12.60	50.85
	p	$< .001$****	$< .001$****
Interaction	$F(2,39)$	4.12	2.74
	p	$< .05$*	0.08

Significant level $p < .05$*, $p < .001$****

Table 6: The result of a post-hoc test in "*before and after*" learning

A response rate	$F(1,39)$	p
all	6.89	$< .05$*
0-degree	13.91	$< .001$****
80-degree	0.04	0.84

Significant level $p < .05$*, $p < .001$****

learning conditions also differ significantly ($F(2,39) = 4.12$, $p < .05$*). Therefore, the interaction should be evaluated. The results of the simple main effect test are shown in Tab.6 and Tab.7. Significant difference is found in "*before and after*" learning within "all" ($F(1,39) = 6.89$, $p < .05$*) and "*0-degree*" ($F(1,39) = 13.91$, $p < .001$****). So the rotation angle seems to affect the response rate in "*all*" and "*0-degree*".

The learning of "*after*" differs significantly ($F(2,78) = 5.61$, $p < .01$**) in the learning conditions. The result of multiple comparisons in the learning conditions of the response rate is shown in Tab.8. The relationship between "*all*" and "*80-degree*" shows significant difference ($t = 3.16$, $p < .01$**), and the relationship between "*0-degree*" and the "*80-degree*" shows significant difference, too ($t = 2.55$, $p < .05$*). Therefore, it turns out that the group of "*all*" gets considerable learning effects rather than "*0-degree*". Namely, the hypothesis is rejected from the fact that the learning flow affects the spatial reasoning capacity. Thus, the effect of the attribute "*all*" and "*0-degree*" is validated.

In the result of the accuracy rate, "*before and after*" learning differs significantly ($F(1,39) = 50.85$, $p < .001$****).

Table 7: The result of a post-hoc test for learning conditions

A response rate	$F(2,78)$	p
before	2.14	0.12
after	5.61	$< .01$**

Significant level $p < .01$**

Table 8: The result of a post-hoc test in "*after*" learning

A response rate			t	p
All	-	*80-degree*	3.16	$< .01$**
All	-	*0-degree*	0.61	0.54
0-degree	-	*80-degree*	2.55	$< .05$*

Significant level $p < .05$*, $p < .01$**

Therefore, the spatial object recognition accuracy has the possibility to be improved regardless of any rotation angle.

As the result of the ANOVA, the response rate is improved by the "*before and after*" AR learning. In this case, both of "*all*" and "*0-degree*" are improved from the view point of the rotation angle effect. Since this method is to have examinees to image object rotation with ninety degrees, "*80-degree*" is too much support for such object rotation in mind. Thus, the rotation angle affects the spatial object recognition speed. Furthermore, the accuracy rate is improved with any conditions and does not differ significantly. It means that the improvement is not affected so much by the rotation angle but the AR learning itself. Therefore, the spatial object recognition accuracy is not affected by the rotation angle.

4.2 Learning with various angles

In this subsection, we perform another experiment to study the effect of the AR learning with rotating the AR marker. The examinees are 16 female students of our university. The experimental procedure is shown as follows.

Step-i) Paper test 1 (5 min.)
Step-ii) Paper test 2 (5 min.)
Step-iii) Rotate the AR marker by θ in a random manner
Step-iv) AR learning (3 min.)
Step-v) Paper test 3 (5 min.)
Step-vi) Terminate, if all angles are selected
Step-vii) Go to step-iii)

In Step-i), the preliminary spatial reasoning capacity is measured using test 1. The test is the same as in section 3. In Step-ii), the spatial reasoning capacity after the test 1 is measured using test 2. In Step-iii), the AR marker is rotated by zero, twenty, forty, sixty and eighty degrees in a random order only once. In Step iv), the AR learning is performed using the method of section 2.3 for 3 minutes. In Step-v), the spatial reasoning capacity after the learning is measured using test 3. In Step-vi), when all the five angles are examined, the procedure is terminated. Otherwise it returns to Step-iii). The total time for the procedure is $5 + 5 + (3 + 5) * 5 = 50$ minutes.

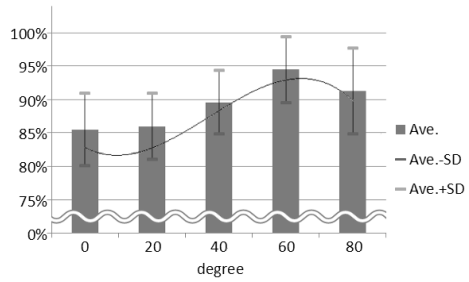


Fig. 13: The relationship between the rotation angle and the accuracy rate

Table 9: Multiple comparisons of angles

Degree			t	p
0	-	80	3.25	$< .005$***
20	-	80	2.98	$< .005$***
40	-	80	0.93	0.36
60	-	80	2.05	$< .05$*

Significant level $p < .05$*, $p < .005$***

We analyze the results of the experiment to investigate which rotation angle is the most effective. To investigate the relationship between the rotation angle and the accuracy rate, a regression analysis is performed. Figure 13 illustrates the average accuracy rate and a regression model [7]. This model is expressed as follows.

$$y = -0.12 * 10^{-5}x^3 + 0.13 * 10^{-3}x^2 - 0.22 * 10^{-2}x + 0.86$$

The effect of learning does not change from zero to twenty degrees. The accuracy rate increases for twenty to sixty degrees. Then the accuracy rate shows a gradual decline for eighty degrees. As pointed out in section 4.1, "*80-degree*" is too much support for such object rotation in mind.

In this experiment, "*80-degree*" improves the accuracy rate rather than "*0-degree*". However, "*0-degree*" improves the accuracy rate rather than "*80-degree*"in section 4.1. This result contradicts the result in section 4.1. Therefore, we examine a one-way ANOVA test. One attribution is the rotation angle. In the results of the one-way ANOVA, the relationship among five rotation angles shows significant difference ($F(4, 56) = 8.99$, $p < .001$****). Therefore, a post-hoc test should be examined. The relationship between "*80-degree*" and other angles in the results of the multiple comparison is shown in Tab.9. There are significant difference between "*80-degree*" and "*0/20-degree*". In the experiment in section 4.1, other angles have no effect with "*80-degree*". However in this experiment, the list of the learning is randomly generated, and the other angles are confirmed to be affected. Since the learning of "*0-degree*" does not support the rotation, the learning itself is difficult. Hence, we conclude that the learning of "*0-degree*" is an unsuitable angle.

As the result of the analysis, it turns out that the AR learning with sixty degrees as the limitation of angle is the most effective method.

5. Conclusions

In this paper, we proposed new methods to train students' spatial reasoning capacity using AR. In the AR environment, virtual objects are displayed even in floating and can be moved freely. Students with low spatial reasoning capacity can effectively learn by rotating an AR marker. In the first method, if the rotation of a virtual object in mind is hard, the AR marker can be rotated by arbitrary angle physically. The second method has physical limitation of the rotation.

To validate the effect of the trained and resultant spatial reasoning capacity, we perform an experiment using the proposed method. As the result of the ANOVA, the accuracy rate is improved by "*before and after*" AR learning. In the case of the accuracy rate, both of "*with frame*" and "*without frame*" are improved while the effect of the box frame is not observed. Thus, the AR learning affects the spatial object recognition accuracy.

To study the relationship between rotation angle and the learning efficiency of examinees, we performed two experiments. First, we performed an experiment to investigate the difference among the cases of fixed angle and variable angles. As the result of the ANOVA, the response rate is improved by the "*before and after*" AR learning. In this case, both of "*all*" and "*0-degree*" are improved from the view point of the rotation angle effect. Second, we performed an experiment to study the effect of the AR learning with rotating the AR marker. The effect of the AR learning does not change from zero to twenty degrees. The accuracy rate increases for forty to sixty degrees. Then the accuracy rate shows a gradual decline for eighty degrees. From the result of the analysis, it turns out that the AR learning with sixty degrees as the limitation of angle is the most effective method.

In our future work, we plan to develop an AR tool to train spatial reasoning capacity for understanding 3D objects from opened-up cubes.

References

[1] the Ministry of Education, government curriculum guidelines, Printing Bureau, Ministry of Finance, 2009
[2] Brett E. Shelton, Copyright 2002 by New Horizons for Learning, Vol.9, 2002
[3] Arduino <http://www.arduino.cc/>
[4] ARToolKit <http://www.hitl.washington.edu/artoolkit/>
[5] OpenGL <http://www.opengl.org/>
[6] Jean Piaget, Journal of Research in Science Teaching, Vol.2, Issue 3, pp.176-186, 1964
[7] I. Gusti Nqurah Agung, Time Series Data Analysis Using EViews (Statistics in Practice), Copyrighted Material, 2009

Queuing Network Approximation Technique for Evaluating Performance of Computer Systems with Multiple Memory Resource Requirements

Afiza Razali *fiza0108@gmail.com*
Toshiyuki Kinoshita *kinoshi@cs.teu.ac.jp*
Akira Tanabe *ddr-br1dxd3t@nifty.com*

Graduate School of Computer Science, Tokyo University of Technology
1404-1 Katakura, Hachioji Tokyo, 192-0982, Japan

Abstract *Queuing network techniques are effective for evaluating the performance of computer systems. We discuss a queuing network technique for computer systems with multiple memory resource requirements. When a job arrives from outside the network, it occupies one of the memory resources and executes CPU and I/O processing in the network occupying the memory. When the job completes the CPU and I/O processing, it releases the memory and leaves the network. However, because the memory resource is considered to be a secondary resource for the CPU and I/O equipment, and because the queuing network model of computer systems with memory resources is an open one, we cannot calculate its exact solutions.*

We propose here an approximation technique for calculating the performance measures of computer systems with multiple memory resource requirements using the queuing network technique. This technique involves dividing a network into two parts; one is a "processing part" in which a job executes CPU and I/O processing, and the other is a "memory part" that indicates how the memory resources are used by jobs. By dividing the network into two parts, we can prevent the number of network states from increasing and can approximately calculate the performance measures of the network. We evaluated the proposed approximation technique using numerical experiments.

Keywords *performance of computer systems, central server model, memory resource, memory resource requirements*

1. Introduction

Queuing network techniques are effective for evaluating the performance of computer systems. In computer systems, two or more jobs are generally executed at the same time, which causes delays due to conflicts in accessing hardware or software resources such as the CPU, I/O equipment, or data files. We can evaluate how this delay affects the computer system performance by using a queuing network technique. Some queuing networks have an explicit exact solution, which is called a product form solution [1]. With this solution, we can easily calculate the performance measures of computer systems, for example the busy ratio of hardware and the job response time. However, when the exclusion controls are active or when a memory resource exists, the queuing network does not have a product form solution. To calculate an exact solution of a queuing network that does not have a product form solution, we have to construct a Markov chain that describes the stochastic characteristics of the queuing network and numerically solve its equilibrium equations. The number of unknown quantities in the equilibrium equations is the same as the number of states of the queuing network. Since the number of states of the queuing network drastically increases when the number of jobs or the amount of hardware in the network increases, the number of unknown quantities in the equilibrium equations also drastically increases. Therefore, we cannot numerically calculate the exact solution of the queuing network. Moreover, when the queuing network is an open model where jobs arrive from or depart for the outside of the network, the number of states of the network can become infinite (the number of jobs can be infinite.), and we cannot actually calculate an exact solution.

We discuss the queuing network technique for computer systems with memory resources. When a job arrives from outside the network, it occupies a portion of the memory resources and executes CPU and I/O processing in the network with the memory. When the job completes CPU and I/O processing, it releases the memory and leaves the network. Therefore, memory can be considered as a secondary resource for the CPU and I/O equipment. When a queuing network includes a secondary resource, it does not have product form solutions. Moreover, since the queuing network technique of computer systems with memory resources is an open model, we cannot calculate its exact solutions.

We propose an approximation technique for calculating the performance measures of computer systems with multiple memory resource requirements. We previously reported the results for a single job type case [6]. In this

paper, we extend the results of [6] and discuss them for the case when there are multiple memory resource requirements and multiple job types (each of which is called a job class). Similarly to the single job class in [6], we divide the network into two parts in order to prevent the number of states of the Markov chain from increasing. One part is called the "processing part," in which jobs execute CPU and I/O processing, and the other is the "memory part," which indicates how the memory resources are used by the jobs. As with the behavior of CPU and I/O processing, the memory usage behavior also differs for each job class, such as how much memory is allocated to the job. When there is a single job class, both the processing and memory parts have a product form solution, but when there are multiple job classes, only the processing part has a product form solution. Therefore, it is not sufficient to divide the queuing network into two parts; an approximation is also needed to analyze the memory part.

Dividing the model into primary and secondary resources is a two-layer queuing network techniques [3][4]. Our proposed technique is also a two-layer technique for computer systems with memory resources.

2. Model Description

The processing part is equivalent to the ordinary central server model with multiple job classes. In this model, K job classes exist, and each of them is numbered k =1, 2, … , K by affixing a k. The processing part consists of a single CPU node and multiple I/O nodes. We denote M as the number of I/O nodes. The I/O nodes are numbered m =1, 2, … , M by affixing m, and the CPU node is numbered m =0 by also affixing m. The service rate of job class k at the CPU node is μ_0^k, and the service rate of job class k at an I/O node m is μ_m^k. The service time at each node is a mutually independent random variable subject to common exponential distributions. Jobs are scheduled on a first come first served (FCFS) principle at all nodes. At the end of CPU processing, a job probabilistically selects a I/O node and moves to it, or completes CPU and I/O processing and departs from the network. The selection probability of I/O node m of job class k is p_m^k (k =1, 2, ... , K; m =1, 2, ... , M), and the completion probability of job class k is p_0^k (k =1, 2, ... , K). Therefore, $\sum_{m=0}^{M} p_m^k = 1$ (k =1, 2, ... , K).

Memory resources are added to this ordinary central server model (Figure 1). We denote S_k as the number of memory resources for job class k. A job from job class k arrives from the outside at random at an arrival rate λ_k. The

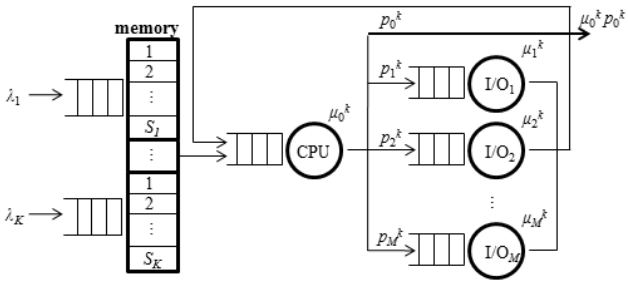


Fig. 1 Central server model with multiple memory resource requirements

job requests and acquires a memory resource before entering the processing part. If all the memory resources for job class k are occupied when the job of job class k arrives, the job joins the memory-waiting queue and waits for one of the memory resources to be released by another job. When the job completes CPU and I/O processing, it releases the memory resource and leaves the network. Because the job has to occupy a memory resource upon entering the processing part, the number of jobs occupying a memory resource is always equal to the number of jobs in the processing part. Therefore, at most S_k jobs of job class k can enter the processing part. That is, the maximum job multiplicity of job class k in the processing part is S_k. The number of jobs of job class k in the processing part is denoted by n_k (= 0, 1, 2, ... , S_k). By replacing "CPU → outside transition" with "CPU → CPU transition," the processing part is modified to a closed central server model in which the number of jobs is constant (Figure 2). In this closed model, when "CPU → CPU transition" occurs, the job terminates and a new job starts. Therefore, the mean job response time is the mean time between two successive "CPU → CPU transitions." This means job response time can be considered as a job lifetime.

3. Approximation Model

To obtain the exact solution of the central server model with memory resource requirements, we have to describe the entire model with a single Markov chain for each job class. However, this causes the number of states of the Markov chain to drastically increase when the number of nodes and the number of jobs in the network increase. By dividing the central server model into two parts (processing part and memory part), and describing each part with two Markov chains, we can prevent the number of states of the model from increasing (Figure 2). We set the following notations.

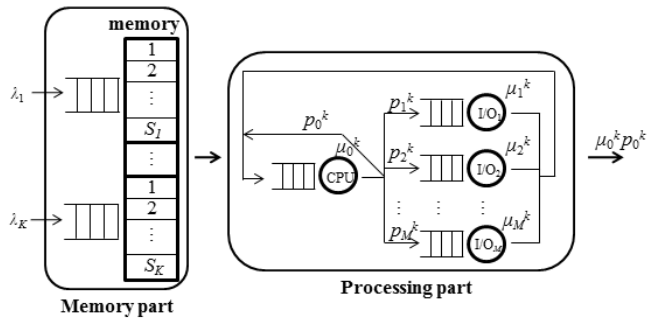


Fig. 2 Concept of approximation

We set the following notations.

τ_{km} : total mean service time at node-m in a job lifetime of job class k (S_k, k=1, 2, ... , K; m =0, 1, ... , M)

$\boldsymbol{n}^* = (n_1, n_2, \dots, n_K)$
: vector of number of jobs (n_k =0, 1, 2, ... , S_k)

n_{km} : number of jobs in job class k at node-m (m =0, 1, ... , M)

$\boldsymbol{n} = (n_{10}, n_{11}, \dots, n_{1M}, n_{20}, n_{21}, \dots, n_{2M}, \dots, n_{K0}, n_{K1}, \dots, n_{KM})$
: state vector of the processing part

$$\boldsymbol{F}(\boldsymbol{n}) = \{\boldsymbol{n} \mid \sum_{m=0}^{M} n_{km} = n_k,\ n_{km} \geq 0 \ (m = 0, 1, \dots, M)\}$$

(n_k =0, 1, 2, ... , S_k, k =1, 2, ... , K)
: set of all feasible states of the processing part when the number of jobs of job class k is n_k

$P_s(\boldsymbol{n})$: steady-state probability of state $\boldsymbol{n}$

The processing part is equivalent to the ordinary central server model with multiple job classes, and therefore, the Markov chain describing the processing part has a product form solution. Then the steady-state probability $P_s(\boldsymbol{n})$ is represented by the following formula [1][2].

$$P_s(\boldsymbol{n}) = \frac{\prod_{k=1}^{K}\prod_{m=0}^{M} \tau_{km}{}^{n_{km}}}{\varphi(n_1, n_2, \cdots, n_K, M)}$$

where $\varphi(n_1, n_2, \cdots, n_K, M) = \sum_{n \in F(n)} \prod_{k=1}^{K}\prod_{m=0}^{M} \tau_{km}{}^{n_{km}}$ is the normalizing constant of steady-state probabilities when the number of jobs of job class k in the processing part is n_k (k =1, 2, ... , K). From these steady-state probabilities, we can calculate the mean job response time $T_{\boldsymbol{n}}{}^k$ of job class k in the processing part, when the number of jobs is n_k (k =1, 2, ... , K), as follows.

$$T_n{}^k = \frac{n_k \cdot \varphi(n_1, \cdots, n_k, \cdots, n_K, M)}{\varphi(n_1, \cdots, n_k - 1, \cdots, n_K, M)}$$

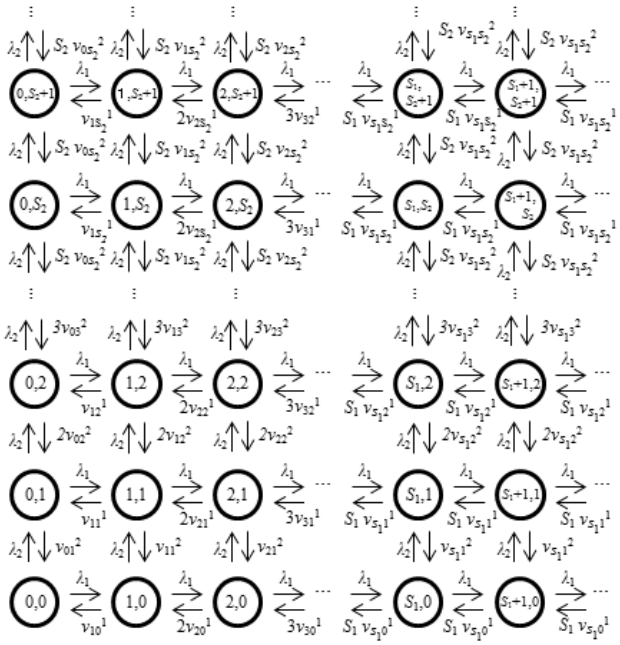


Fig. 3 State Transition diagram (two job classes)

The memory part can be considered as an M/M/S_k queuing model with S_k servers (k =1, 2, ... , K). In an ordinary M/M/S_k queuing model, the service rate at a server is constant, regardless of the number of guests in the service. In the memory part, however, the service rate changes depending on the number of occupied memory resources. The mean job response time $T_{\boldsymbol{n}}{}^k$ of job class k (k =1, 2, ... , K) in the processing part, when the number of jobs in the processing part is $\boldsymbol{n}^* = (n_1, n_2, \dots, n_K)$, is equal to the mean time of occupied memory. Since the service rate from the processing part $v_{\boldsymbol{n}}{}^k$ is denoted as $v_{\boldsymbol{n}}{}^k = \frac{1}{T_{\boldsymbol{n}}{}^k}$, $v_{\boldsymbol{n}}{}^k$ depends on the number of occupied memory resources n_k. The state transition of the M/M/S_k queuing model (k =1, 2) is shown in Figure 3, where the memory service rates change depending on the number of occupied memory resources. This is a two-dimensional birth-death process. The equilibrium equations with the steady-state probability $Q_{\boldsymbol{S}}(\boldsymbol{n}^*) = Q_{\boldsymbol{S}}(n_1, n_2)$, when the number of memory resources is $\boldsymbol{S} = (S_1, S_2)$ and the number of occupied memory resources is $\boldsymbol{n}^* = (n_1, n_2)$, are as follows (similar to the case with higher dimensions).

$$(\lambda_1 + \lambda_2) \cdot Q_{\boldsymbol{S}}(0, 0) = v_{10}{}^1 \cdot Q_{\boldsymbol{S}}(1, 0) + v_{01}{}^2 \cdot Q_{\boldsymbol{S}}(0, 1)$$

$$(\lambda_1 + \lambda_2 + n_1 v^1_{n_1 0}) \cdot Q_{\boldsymbol{S}}(n_1, 0) = \lambda_1 \cdot Q_S(n_1 - 1, 0) + (n_1 + 1)\, v^1_{n_1+1\,0} \cdot Q_{\boldsymbol{S}}(n_1 + 1, 0) + v^2_{n_1 1} \cdot Q_{\boldsymbol{S}}(n_1, 1)$$

(n_1=1, 2, ... , $S_1 - 1$)

$$(\lambda_1+\lambda_2+ S_1\nu^1_{S_1 0})\cdot Q_S(n_1, 0) = \lambda_1\cdot Q_S(n_1-1, 0) + S_1\nu^1_{S_1 0}\cdot Q_S(n_1+1, 0) + \nu^2_{n_1 1}\cdot Q_S(n_1, 1) \quad (n_1= S_1, S_1+1, ...)$$

$$(\lambda_1+\lambda_2+n_2\nu^2_{0n_2})\cdot Q_S(0, n_2) = \lambda_2\cdot Q_S(0, n_2-1) + \nu^1_{1n_2}\cdot Q_S(1, n_2) + (n_2+1)\nu^2_{0n_2+1}\cdot Q_S(0, n_2+1) \quad (n_2 =1, 2, ..., S_2-1)$$

$$(\lambda_1+\lambda_2+S_2\nu^2_{0n_2})\cdot Q_S(0, n_2) = \lambda_2\cdot Q_S(0, n_2-1) + \nu^1_{1n_1}\cdot Q_S(1, n_2) + S_2\nu^2_{0S_2}\cdot Q_S(0, n_2+1) \quad (n_2 =S_2, S_2+1, ...)$$

$$(\lambda_1+\lambda_2+n_1\nu^1_{n_1n_2}+n_2\nu^2_{n_1n_2})\cdot Q_S(n_1, n_2) = \lambda_1\cdot Q_S(n_1-1, 0) + \lambda_2\cdot Q_S(0, n_2-1) + (n_1+1)\nu^1_{n_1+1n_2}\cdot Q_S(n_1+1, n_2) + (n_2+1)\nu^2_{n_1n_2+1}\cdot Q_S(n_1, n_2+1) \quad (n_1 =1, 2, ..., S_1-1;\ n_2 =1, 2, ..., S_2-1)$$

$$(\lambda_1+\lambda_2+S_1\nu^1_{S_1n_2}+n_2\nu^2_{n_1S_2})\cdot Q_S(n_1, n_2) = \lambda_1\cdot Q_S(n_1-1, 0) + \lambda_2\cdot Q_S(0, n_2-1) + S_1\nu^1_{S_1n_2}\cdot Q_S(n_1+1, n_2) + (n_2+1)\nu^2_{n_1n_2+1}\cdot Q_S(n_1, n_2+1) \quad (n_1 =S_1, S_1+1, ...;\ n =1, 2, ..., S_2-1)$$

$$(\lambda_1+\lambda_2+n_1\nu^1_{n_1n_2}+S_2\nu^2_{n_1S_2})\cdot Q_S(n_1, n_2) = \lambda_1\cdot Q_S(n_1-1, n_2) + \lambda_2\cdot Q_S(n_1, n_2-1) + (n_1+1)\nu^1_{n_1+1n_2}\cdot Q_S(n_1+1, n_2) + S_2\nu^2_{n_1S_2}\cdot Q_S(n_1, n_2+1) \quad (n_1 =1, 2, ..., S_1-1;\ n_2=S_2, S_2+1, ...)$$

$$(\lambda_1+\lambda_2+S_1\nu^1_{S_1n_2}+S_2\nu^2_{n_1S_2})\cdot Q_S(n_1, n_2) = \lambda_1\cdot Q_S(n_1-1, n_2) + \lambda_2\cdot Q_S(n_1, n_2-1) + S_1\nu^1_{S_1n_2}\cdot Q_S(n_1+1, n_2) + S_2\nu^2_{n_1S_2}\cdot Q_S(n_1, n_2+1) \quad (n_1 =S_1, S_1+1, ...;\ n_2 =S_2, S_2+1, ...)$$

The transition diagram of the two-dimensional birth-death process is shown in Figure 3. However, the steady-state equation does not have a product form solution. Therefore, some approximation is required to solve it.

When a single job class and a single memory resource exist in the model, it can be described with a one-dimensional birth-death process. Its transition diagram is shown in Figure 4, and the steady-state equation is as follows:

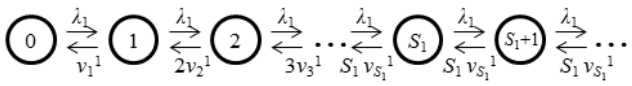


Fig. 4 State Transition diagram (single job class)

$$\lambda_1\cdot Q_S(0)=\nu_1{}^1\cdot Q_S(1)$$

$$(\lambda_1+ n_1\ \nu^1_{n_1})\cdot Q_S(n_1) =\lambda_1\cdot Q_S(n_1-1) + (n_1+1)\ \nu^1_{n_1+1}\cdot Q_S(n_1+1) \quad (n_1 =1, 2, ..., S_1-1)$$

$$(\lambda_1+S_1\nu^1_{S_1})\cdot Q_S(n_1) =\lambda_1\cdot Q_S(n_1-1)+) + S_1\nu^1_{S_1}\cdot Q_S(n_1+1) \quad (n_1 =S_1, S_1+1, ...)$$

Solutions for the steady-state equation are in the following product form.

$$Q_S(n_1) = Q_S(0)\cdot\frac{1}{n_1!}\prod_{i=1}^{n_1}\left(\frac{\lambda_1}{\nu^1_i}\right) \quad (n_1=1, 2, ..., S_1-1)$$

$$= Q_S(0)\cdot\frac{1}{S_1!S_1^{\,n_1-S_1}}\prod_{i=1}^{S_1}\left(\frac{\lambda_1}{\nu^1_i}\right)\cdot\left(\frac{\lambda_1}{\nu^1_{S_1}}\right)^{n_1-S_1} \quad (n_1=S_1, S_1+1, ...)$$

In this formula, for the state transition of i =1, 2, ... , S_1-1, multiply by factor $\frac{\lambda_1}{i\cdot\nu^1_i}$ while for the state transition of i = S_1, S_1+1, ... , multiply by factor $\frac{\lambda_1}{S_1\cdot\nu^1_{S_1}}$. For two-dimension case, we consider routes from lattice point (0, 0) to (n_1, n_2) shown in Figure 3, and for the state transition to the right direction along a route, multiply by factor $\frac{\lambda_1}{i\cdot\nu^1_i}$ (i =1, 2, ... , S_1-1) or $\frac{\lambda_1}{S_1\cdot\nu^1_{S_1}}$ (i =S_1, S_1+1, ...), while for the state transition in the upward direction, multiply by factor $\frac{\lambda_2}{j\cdot\nu^2_j}$ (j =1, 2, ... , S_2-1) or $\frac{\lambda_2}{S_2\cdot\nu^2_{S_2}}$ (j =S_2, S_2+1, ...). Thus, the coefficient of $Q_S(n_1, n_2)$ related to $Q_S(0, 0)$ is represented as the multiplication based on the route from (0, 0) to (n_1, n_2). Since there are multiple routes from (0, 0) to (n_1, n_2), the

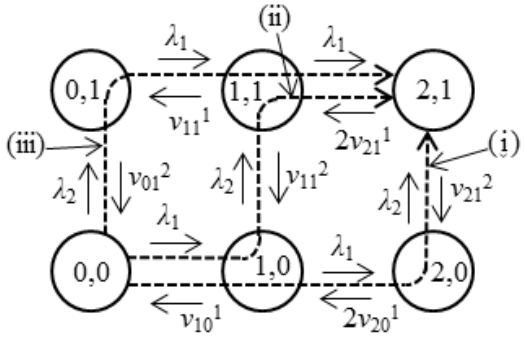


Fig. 5 Calculating State Probability along routes to (2, 1)

coefficient of $Q_S(n_1, n_2)$ related to $Q_S(0, 0)$ is approximately represented as the average of the multiplication based on all routes. For example, since there are three routes from $Q_S(0, 0)$ to $Q_S(2, 1)$ then,

$$Q_S(2,1) = Q_S(0,0) \times \frac{1}{3} \cdot \left\{ \left(\frac{\lambda_1}{\nu_{10}^1}\right)\left(\frac{\lambda_1}{2\cdot\nu_{20}^1}\right)\left(\frac{\lambda_2}{\nu_{21}^2}\right) \right. \quad \text{..... (i)}$$

$$+ \left(\frac{\lambda_1}{\nu_{10}^1}\right)\left(\frac{\lambda_2}{\nu_{11}^2}\right)\left(\frac{\lambda_1}{2\cdot\nu_{21}^1}\right) \quad \text{.... (ii)}$$

$$\left. + \left(\frac{\lambda_2}{\nu_{01}^2}\right)\left(\frac{\lambda_1}{\nu_{11}^1}\right)\left(\frac{\lambda_1}{2\cdot\nu_{21}^1}\right) \right\} \quad \text{.... (iii)}$$

Here (i)(ii)(iii) are the terms that were calculated on the basis of the corresponding routes of the broken lines in Figure 5. Similarly to the case above, we can approximately calculate the state probability of a queuing network with multiple memory resource requirements when K >2.

4. Numerical Experiments

We evaluated the proposed approximation technique through numerical experiments. We used the following parameters.

1. Number of memory resources: (S_1, S_2)=(2, 2), (3,3), (4, 4)
2. Arrival rate: λ_1=0.0 -- 21.0, λ_2 =1.0, 4.0
3. Number of I/O nodes: M =2
4. Total service time at each node
   τ_{10}=1.0, τ_{11}=τ_{12}=0.5
   τ_{20}=1.0, τ_{21}=τ_{22}=1.0
   where τ_{km} is the total service time of job class k at node m.

Figures 6 and 7 show the mean job response times of job classes 1 and 2 as T_1, T_2 respectively, when λ_2 is fixed at 1.0 and 4.0, and λ_1 changes from 0.0 to 21.0. In these graphs, T_k (S_1, S_2) is the mean job response time of job class k when the numbers of memory resources of job classes 1and 2 are S_1, S_2 respectively. Similarly to the case of a single job class, the mean job response time for both job classes increases monotonically in a convex curve. As the arrival rate λ_1 increases and λ_2 is fixed (the load of only job class 1 increases), the mean response time of both job classes 1 and 2 increases. We can see that the mean response time of job class 2 is increasing higher than that of job class 1, and the reason for this is presumed to be that job class 2 has a longer I/O time than job class 1.

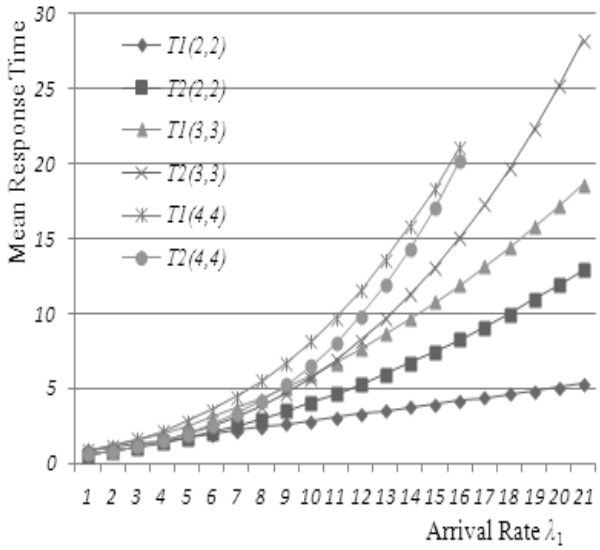


Fig. 6 Mean Job Response Time (λ_2=1.0)

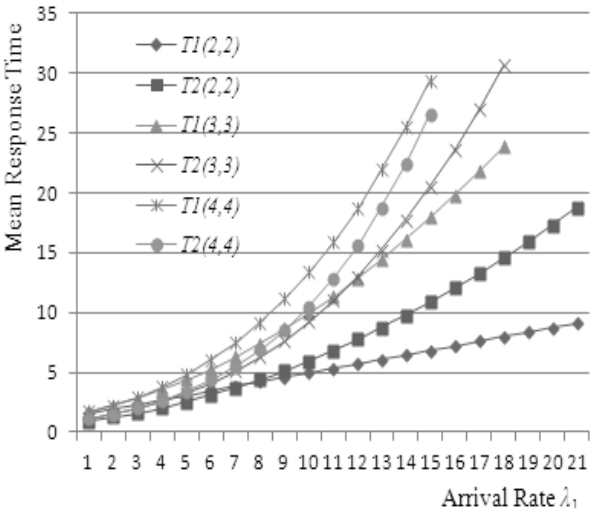


Fig. 7 Mean Job Response Time (λ_2=4.0)

5. Conclusion

We proposed an approximation technique for evaluating the performance of computer systems with multiple memory resource requirements using a queuing network and analyzed its performance measures through numerical experiments. The concept of the approximation is based on separately analyzing the processing part and the memory part of the queuing network model.

The numerical experiments clarified the characteristics of the mean job response time.
In the future we plan to examine the accuracy of the proposed approximation technique by comparing it with exact solutions or simulation results.

REFERENCES

[1] F. Baskett, K. M. Chandy, R. R. Muntz and F. G. Palacious, “Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," J. ACM, Vol.22, No.2, pp.248--260, April 1975.

[2] H. Kobayashi, “Modeling and Analysis," Addison-Wesley Publishing Company, Inc. 1978.

[3] T. Kurasugi and I. Kino, “Approximation Method for Two-layer Queueing Models," Performance Evaluation 36--37, pp.55--70, 1999.

[4] J. A. Rolia and K. C. Sevcik, “The Method of Layers," IEEE Trans. on Software Engineering, Vol.21, No.8, pp.689--700, Aug. 1995.

[5] T. Kinoshita and Y. Takahashi, “A Queuing Network Modeling and Performance Evaluation Method for Computer Systems with Resource Requirement," IEICE D-I, Vol. J 82-D-I, No.6, pp.701--710, Jun. 1999.

[6] T. Kinoshita and X. Gao, “Queuing Network Approximation Technique for Evaluating Performance of Computer Systems with Memory Resources,” PDPTA2010, pp.640, July 2010

GPU Acceleration of BCP Procedure for SAT Algorithms

Hironori Fujii[1] **and Noriyuki Fujimoto**[1]
[1]Graduate School of Science Osaka Prefecture University
1-1 Gakuencho, Nakaku, Sakai, Osaka 599-8531, Japan

Abstract— *The satisfiability problem (SAT) is widely applicable and one of the most basic NP-complete problems. This problem has been required to be solved as fast as possible because of its significance, but it takes exponential time in the worst case to solve. Therefore, we aim to save the computation time by parallel computing on a GPU. We propose parallelization of BCP (Boolean Constraint Propagation) procedure, one of the most effective techniques for SAT, on a GPU. For a 2.93GHz Intel Core i3 CPU and an NVIDIA GeForce GTX480, our experiment shows that the GPU accelerates our SAT solver based on our BCP-embedded divide and conquer algorithm 6.7 times faster than the CPU counterpart*

Keywords: Satisfiability problem, Boolean constraint propagation, GPGPU, CUDA

1. Introduction

The satisfiability problem (SAT for short) [1] is a problem of determining if a given boolean expression (usually in the conjunctive normal form) can be true by assigning some boolean values into the boolean variables in the expression. Formally, SAT is defined as follows:

- Let $V = \{V_1, V_2, \cdots, V_n\}$ be a set of n boolean variables.
- Let $C = \{C_1, C_2, \cdots, C_m\}$ be a set of m *clauses* where
  - $C_j = (L_{j_1} \vee L_{j_2} \vee \cdots \vee L_{j_{k_j}})$
  - $L_i \in \{V_p, \neg V_p\}$
- Question: Is there assignment to the variables in V such that $C_1 \wedge C_2 \wedge \cdots \wedge C_m = \text{true}$?

where $\neg$ is a logical not operator, $\vee$ is a logical or operator, $\wedge$ is a logical and operator. L_i is called a *literal* which is either affirmation or negation of a boolean variable. SAT problem such that clauses are of exactly three literals is called 3SAT. Any SAT instance can be transformed to a 3SAT instance with the same solution. Therefore, without loss of generality, for simplicity, our SAT solver accepts 3SAT instance only.

SAT is widely applicable and one of the most basic NP-complete problems. This problem has been required to be solved as fast as possible because of its significance, but it takes exponential time in the worst case to solve. Recent SAT algorithms can solve problem instances with several million variables in a few hours. SAT algorithms can be categorized into complete-type and incomplete-type. Complete-type algorithms can identify whether any given problem instance is satisfiable or unsatisfiable. In contrast, incomplete-type algorithms can not identify unsatisfiability but satisfiability. Many of common complete-type algorithms can be categorized into so-called DPLL algorithm [2], [3]. DPLL algorithm performs BCP(Boolean Constraint Propagation) procedure, which is a dominant part of the algorithm. BCP procedure occupies 80% to 90% of the whole execution. On the other hand, Davis et al. proposed a method [4] to accelerate BCP procedure by parallel processing with an FPGA. Davis et al. parallelized only BCP procedure for their method to be applicable to various DPLL algorithms. With Xilinx Virtex 5 LX110T and 3.6GHz Intel Pentium 4, Davis et al. experimentally showed that zChaff [5], [6], [7] with FPGA accelerated BCP procedure runs 5 to 16 times faster than the zChaff with the CPU only.

This paper proposes parallelization of BCP procedure on a GPU, rather than an FPGA. Similar to Davis et al.'s method, the proposed method can be used with various DPLL algorithms. However, we adopt divide-and-conquer algorithm [14] (3SAT-DC for short) with BCP procedure as our SAT solver used in the experiments in this paper. 3SAT-DC is complete-type. For a 2.93GHz Intel Core i3 CPU and an NVIDIA GeForce GTX480, our experiment showed that the GPU accelerates our SAT solver based on our BCP-embedded divide and conquer algorithm 6.7 times faster than the corresponding CPU implementation.

The remainder of this paper is organized as follows. Section 2 briefly reviews DPLL algorithm, BCP procedure, and SAT-DC. Section 3 presents the proposed algorithm. Experiments to show the performance of the proposed algorithm are reported in Section 4. Section 5 briefly surveys the related works. Section 6 gives some concluding remarks and future works. Due to the limited space, this paper include no description on CUDA. Readers unfamiliar with CUDA GPU architecture are recommended the literature [8], [9], [10], [11], [12], [13].

2. Preliminaries

2.1 DPLL Algorithm

Basically, DPLL algorithm finds an optimal solution by checking all the patterns of boolean value assignment to the variables in a given boolean expression while discarding hopeless sets of patterns without checking them. One of the methods to identify a hopeless set of patterns is

BCP procedure. Listing 1 shows a pseudo code of DPLL algorithm with BCP procedure. DPLL algorithm performs BCP procedure every *decision phase* which heuristically determines the value of a variable. Other features of DPLL algorithm are not used in the proposed method, and therefore not described in this paper. See the literatures [2], [3] for more detail of DPLL algorithm.

2.2 BCP Procedure

BCP procedure consists of "implication" process and "conflict" process. The *"implication" process* finds a literal whose value is consequently determined by other literals in the same clause, and then repeat this process until no such a literal is found. The *"conflict" process* finds a clause with all literals false. Listing 2 shows a C code of BCP procedure. In the following C codes, literals and clauses are respectively stored in arrays named "atom" and "clause". The number of literals and clauses are hold in variables named "numatom" and "numclause". Value of a literal is true, false, or unknown, which are represented as 1, -1, and 0 respectively. BCP procedure repeatedly calls function BCPEngine (line 7 in Listing 2) until no more implication occurs or conflict is detected. Listing 3 shows a C code of function BCPEngine. After "implication" process, BCPEngine scans all clauses to detect implication or conflict in $O(m)$. If every literal in a clause is false, then BCPEngine finishes after updating flag variable "conf" to indicate that conflict occurs (line 5 to 8 in Listing 3). Otherwise, BCPEngine performs "implication" process. If exactly one literal is unknown (line 11 to 13 in Listing 3) and any other literal is false, then BCPEngine detects implication and assigns the value which makes the unknown literal true into the corresponding element of array "atom" (line 20 to 22 in Listing 3). Then, BCPEngine pushes the index value of the modified element of array "atom" to array "implicated" and increments the stack pointer "sp" (line 23 in Listing 3). Finally, BCPEngine updates flag variable "imp" to indicate the detection of implication (line 24 in Listing 3).

2.3 3SAT-DC

Listing 4 shows a pseudo code of 3SAT-DC where the parameter "f" of function "3SAT-DC" is a given logical expression and notation "f(x=propositional constant)" represents the logical expression such that "f" is simplified by substituting the constant to propositional variable "x" . 3SAT-DC is a recursive procedure for a clause with the minimum unknown literals at that time. The time complexity of 3SAT-DC is $O(m1.84^n)$ [14].

3. The Proposed Algorithm

3.1 An Overview

BCP procedure is inherently sequential because it consists of iterations of $O(m)$ time. Therefore, we parallelize only each iteration on a GPU and performs the other part on a CPU. To do so, we partition m clauses. Listing 5 shows a pseudo code of the proposed CPU code for DPLL algorithm with parallelized BCP procedure. After partitioning given clauses, DPLL algorithm with parallelized BCP procedure searches a solution while transferring data on the current status of search to memory on a CPU or a GPU every decision. The difference between serial DPLL algorithm in Section 2.1 and the parallel version is addition of clause partitioning in line 2 of Listing 5.

3.2 Clause Partitioning

We partition clauses into groups in order to parallelize BCP procedure. Each group is processed by a thread block. Hence, we partition clauses into groups the same number of clauses for load balance. The maximum number of active thread blocks is eight times the number numMP of multiprocessors. Hence, we set the number numsubclause of clauses in a group at (numclause + numMP * 8 - 1) / (numMP * 8). Due to this, the number of groups is at most the maximum number of active thread blocks.

3.3 Parallelized BCP Procedure

This section describes how to parallelize the serial BCP procedure in Section 2.2.

3.3.1 Calling from a CPU

Listing 6 shows a CUDA C code of the proposed parallelized BCP procedure called from line 5 in Listing 5. The differences between the serial BCP in Listing 2 and the parallel version are:

- Status data are packed into the same array as many as possible.
- Data transfers between a CPU and a GPU are inserted before and after do-while statement and kernel function call.

The reason why status data are packed into the same array is to reduce the number of cudaMemcpy execution. The overhead of cudaMemcpy invocation is heavy. Therefore, memory transfer time can be reduced by reduce the nuber of the invocations. In the proposed CUDA C code, flag variables "conf" and "imp" are packed into array "flag". Also, stack pointer "sp" and array "implicated" are packed into array "sp_implicated". Due to these packings, the whole execution time is reduced to about 90%. The proposed CUDA C code transfers array "atom" and "sp_implicated" before and after executing parallelized BCP procedure (line 7 to 10 and 18 to 21 in Listing 6). However, if no conflict occurs, the latter transfer is not performed to avoid excess transfer. After executing kernel function BCPEngine, variable "flag" is transfered (line 15 in Listing 6).

Listing 1: A pseudo code, focusing on BCP procedure, of DPLL algorithm

```
preprocess to detect trivial unsatisfiability
while(1){
  Decision
  while (BCP() == conflict) { if (no more backtrack) return FALSE; backtrack }
}
```

Listing 2: Serial BCP procedure

```
int BCP(int numclause, int (*clause)[3], int *atom)
{
  int conf = 0, imp;
  do {
    imp = 0; BCPEngine(numclause, clause, atom, &conf, &imp);
  } while (!conf && imp);
  return conf;
}
```

Listing 3: Serial BCP Engine

```
void BCPEngine(int numclause, int (*clause)[3], int * atom, int *conf, int *imp)
{
  *conf = 0; *imp = 0;
  for (int i = 0; i < m; i++) {
    if (clause[i][0] * atom[abs(clause[i][0])] >= 0) goto F;
    if (clause[i][1] * atom[abs(clause[i][1])] >= 0) goto F;
    if (clause[i][2] * atom[abs(clause[i][2])] >= 0) goto F;
    *conf = 1; return;
F:
    int numUnknown = 0; int idxUnknown;
    if (atom[abs(clause[i][0])] == 0) { numUnknown++; idxUnknown = 0; }
    if (atom[abs(clause[i][1])] == 0) { numUnknown++; idxUnknown = 1; }
    if (atom[abs(clause[i][2])] == 0) { numUnknown++; idxUnknown = 2; }
    if (numUnknown == 1) {
      // There is exactly one literal of unknown value in clause[i]
      if (clause[i][0] * atom[abs(clause[i][0])] > 0) continue;
      if (clause[i][1] * atom[abs(clause[i][1])] > 0) continue;
      if (clause[i][2] * atom[abs(clause[i][2])] > 0) continue;
      // Any other literal is false
      int litUnknown = clause[i][idxUnknown];
      int val = (litUnknown > 0) ? TRUE : FALSE;
      atom[abs(litUnknown)] = val;
      implicated[sp++] = abs(litUnknown);
      (*imp)++;
}}}
```

Listing 4: A pseudo code of 3SAT-DC

```
3SAT–DC(f)
{
  if (f == FALSE) return FALSE;
  min_c = a clause with minimum number of literals;
  if( min_c == an empty clause ) return TRUE;
  if( min_c == (x) ) return 3SAT–DC( f(x = true) );
  else if( min_c == (x or y) )
    return 3SAT–DC( f(x = true) ) || 3SAT–DC( f(x = false, y = true));
  else /* min_c==( x or y or z ) */
    return 3SAT–DC( f(x = true) ) || 3SAT–DC( f(x = false, y = true))
             || 3SAT–DC( f(x = false, y = false, z = true));
}
```

Listing 5: A pseudo code of the proposed CPU code for DPLL algorithm with parallelized BCP procedure

```
1 preprocess to detect trivial unsatisfiability
2 partition clauses into groups
3 while(1){
4   Decision
5   while (BCP() == conflict) {
6     if (no more backtrack) return FALSE
7     backtrack
8   }
9 }
```

Listing 6: A CUDA C code of the proposed parallelized BCP procedure

```
1  #define BLKSZ 64
2  int BCP_GPU(int numatom, int *atom, int s, int *sp_implicated, int *d_subset,
3              int *d_orderedClause, int *d_atom, int *d_flag, int *d_sp_implicated)
4  {
5    int flag[2];  // flag[0]:conf, flag[1]:imp
6    cudaMemset(&d_flag[0], 0, sizeof(int));  // conf = 0;
7    cudaMemcpy(d_atom, atom, sizeof(int) * (numatom + 1),
8               cudaMemcpyHostToDevice);
9    cudaMemcpy(d_sp_implicated, sp_implicated, sizeof(int) * (numatom+1),
10             cudaMemcpyHostToDevice);
11   do {
12     cudaMemset(&d_flag[1], 0, sizeof(int));  // imp = 0;
13     BCPEngine<<< s, BLKSZ >>>( d_subset, d_orderedClause,
14                                d_atom, d_flag,d_sp_implicated);
15     cudaMemcpy(flag, d_flag, sizeof(int)*2, cudaMemcpyDeviceToHost);
16   } while (!flag[0] && flag[1]);  // conf == 0 && imp != 0
17   if(!flag[0]){  // no conflict
18      cudaMemcpy(atom, d_atom, sizeof(int) * (numatom + 1),
19                 cudaMemcpyDeviceToHost);
20      cudaMemcpy(sp_implicated, d_sp_implicated, sizeof(int) * (numatom+1),
21                 cudaMemcpyDeviceToHost);
22   }
23   return flag[0];  //return conf;
24 }
```

3.4 Kernel Function BCPEngine

Listing 7 shows the CUDA C code of the proposed kernel function BCPEngine. The number of thread blocks is the number s of groups. The number of threads in a thread block is predefined constant BLKSZ. The arguments subset and orderedClause represent partitioned groups together. Each element of orderedClause holds a clause. Clauses in the same group are consecutively stored in array orderedClause. The head index of elements of each group is stored array "subset". The type of orderedClause is int[4] rather than int[3]. Since every clause of 3SAT instance has exactly three literals, int[3] is sufficient to hold each clause. However, the proposed kernel stores each clause into int[4] with padding in order to enable coalesce access.

The kernel function BCPEngine works as follows. As preparation, the number "size" of clauses in the group assigned to each thread block is calculated (line 4 in Listing 7). Also, pointer "orderedClause" is moved so as to point at the head clause of the assigned group (line 5 in Listing 7). Hence, in the lines below line 5, each thread block can access the assigned group via orderedClause[0..size-1]. The processing for each clause is almost same as the serial BCPEngine in Listing 3, but there are four different points. First, flag "conflict" is checked before each clause is processed and if conflict is detected then each thread is finished (line 8 in Listing 3). Second, each clause is loaded into variable "c" of type int4 to realize coalesced access (line 10 in Listing 3). Third, detection of simultaneous implication by multiple threads is processed by CUDA atomic functions (line 27 to 30 in Listing 7). If multiple threads detect implications for the same variable and attempt to assign the variable different values, then conflict should be detected as shown in line 31 in Listing 7. However, even if line 31 is deleted, no problem occurs because it will be detected as conflict at the next invocation of BCPEngine. Preliminary experiments show that the code without line 31 is faster by a few percent. Therefore, we deleted line 31. Fourth, implication flag variable on global memory is not updated immediately. Instead, each thread updates myImplication

flag variable on a register at each iteration and finally updates implication flag variable once (line 34 in Listing 7).

3.4.1 Cache Configuration

The proposed kernel function in Listing 7 never use shared memory. Instead, it relies on cache memory of Fermi architecture. Fermi GPUs can configure size of shared memory and L1 cache memory either 16KB/48KB or 48KB/16KB. Our current implementation never use shared memory. Therefore, we set the size of L1 cache memory at 48KB.

4. Experiments

This section compares the performance of the proposed CUDA program with a CPU program that performs the same computation. We measured the execution time (average of 10 trials for each test) of not only BCP procedure but also whole execution of 3SAT-DC with BCP procedure. We embedded invocations of BCP procedure just before recursive calls in line 6, 8, 10, and 11 in Listing 4, which correspond to Decisions. Furthermore, we set up the maximum number BCPMAX of invocations of BCP procedure. If our SAT solver executed BCP procedure BCPMAX times, we stopped our SAT solver and measured the execution time at that time. If our SAT solver found a solution or exhausted the search space before BCP procedure was executed BCPMAX times, the execution time at that time is measured. We fixed the number of threads in a thread block at 64 because preliminary experiments showed the number is better.

For each test, a single core of 2.93 GHz Intel Core i3 and NVIDIA GeForce GTX480 was used. The OS used is Windows7 Professional SP1 with NVIDIA graphics driver Version 285.62. For compilation, Microsoft Visual Studio 2008 Professional Edition with optimization option /O2 and CUDA 3.2 SDK were used.

4.1 Performance and Problem Size

In this section, we compare the performance of GPU with that of CPU for various problem sizes. The used problem instances were generated by Motoki's 3SAT instance generator G3 (n,m) [15], [16] (G3 for short). G3 generates a boolean expression that has exactly one solution with high probability. In general, to solve 3SAT instance with many (a few) solutions is easy (hard).

Figure 1, 2, and 3 show a performance comparison between CPU and GPU with problem instances generated by G3. As for Figure 1 and 2, x-axis is the number of variables and y-axis is the execution time in second. Figure 1 shows performance for relatively small instances with BCPMAX 50000. Figure 2 shows performance for relatively large instances with BCPMAX 10000. Figure 3 shows the speedup ratios in case of Figure 1 and 2. As for Figure 3, x-axis is the number of variables and y-axis is speedup ratio. Our GPU solver runs faster with an increase of the problem size (i.e., the number of variables). Although GPU is slower than CPU for small problems, it is reversed for 2500 variables. The best GPU performance is about 4.5 times than CPU for 50000 variables. For more than 50000 variables, the performance of GPU tends to decrease with an increase of the problem size.

4.2 Performance and the Number of BCP Procedure Done

In this section, we fix problem size and compare the performance of GPU with that of CPU for various values of BCPMAX. We randomly selected 10 instances from SAT11 Competition [17] with category RANDOM, 50000 variables, and 210000 clauses.

Figure 4 shows the result. The x-axis shows the name of instance file name and the y-axis is speedup ratio. The speedup ratio is improved with an increase of BCPMAX. Even if BCPMAX is 1000, GPU is 1.5 times faster than CPU in average. The best performance (6.7 times speedup) of GPU is obtained when BCPMAX is 10000. For BCPMAX larger than 10000, the performance improvement is negligible.

5. Related Works

As far as we know, there exist five existing research on SAT algorithm parallelized on a CUDA GPU, as shown below.

In [18], Meyer et al. proposed a complete-type method to parallelize 3SAT-DC like us. However, they did not use BCP procedure. Instead, they proposed a problem partitioning method to parallelize divide-and-conquer itself and their own heuristic for Decision phase.

In [19], McDonald et al. proposed an incomplete-type method to parallelize WalkSAT algorithm with clause learning. Their parallelization method is to run many threads such that each thread executes serial WalkSAT algorithm with pseudo random sequence different any other thread.

In [20], Gulati et al. proposed a complete-type method named MESP (MiniSAT enhanced with SurveyPropagation). They experimentally showed speedup ratio of 2.35 in average with 2.67GHz Intel i7 CPU and NVIDIA GeForce 280GTX GPU.

In [21], Wang et al. proposed an incomplete-type method of a celler genetic algorithm with random walk local search.

In [22], Deleau et al. proposed an incomplete-type method such that SAT instance is represented by a 0-1 matrix and 0-1 matrix multiplication is used to search a solution for a given SAT instance.

6. Conclusion and Future Work

This paper has proposed a method to parallelize BCP procedure for SAT algorithm and has implemented a CUDA GPU program based on the proposed method. Experimental

Listing 7: A CUDA C code of the proposed parallelized BCP procedure

```
__global__ void BCPEngine(int *subset, int (*orderedClause)[4],
                          int *atom, int *flag, int *sp_implicated)
{
   int size = subset[blockIdx.x + 1] - subset[blockIdx.x];
   orderedClause += subset[blockIdx.x];
   int myImplication = 0;
   for (int i = threadIdx.x; i  < size; i += blockDim.x) {
     if (flag[0]) return;  // Another thread detected conflict
     int c[4];
     *((int4 *) c) = *((int4 *) orderedClause[i]);
     if (c[0] * atom[abs(c[0])] >= 0) goto F;
     if (c[1] * atom[abs(c[1])] >= 0) goto F;
     if (c[2] * atom[abs(c[2])] >= 0) goto F;
     flag[0] = 1;  // conf = 1;
     return;
F:
     int numUnknown = 0; int idxUnknown;
     if (atom[abs(c[0])]==UNKNOWN) {numUnknown++; idxUnknown=0;}
     if (atom[abs(c[1])]==UNKNOWN) {numUnknown++; idxUnknown=1;}
     if (atom[abs(c[2])]==UNKNOWN) {numUnknown++; idxUnknown=2;}
     if (numUnknown == 1) {  // exactly one literal of unknown value in orderedClause[i]
       if (c[0] * atom[abs(c[0])] > 0) continue;
       if (c[1] * atom[abs(c[1])] > 0) continue;
       if (c[2] * atom[abs(c[2])] > 0) continue;
       int litUnknown = c[idxUnknown];
       int val = (litUnknown > 0) ? TRUE : FALSE;
       int old = atomicCAS(&atom[abs(litUnknown)], UNKNOWN, val);
       if (old == UNKNOWN) {myImplication++;
                            sp_implicated[atomicAdd(&sp_implicated[0],1)]
                            = abs(litUnknown);}
       //else if (old != val) { flag[0] = 1; return; }
       }
     }
     if(myImplication) flag[1] = 1;  //  imp = 1;
  }
}
```

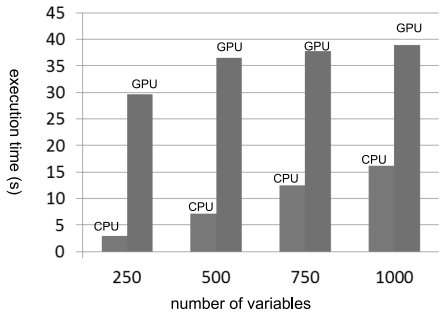


Fig. 1: A performance comparison between CPU and GPU (small instance)

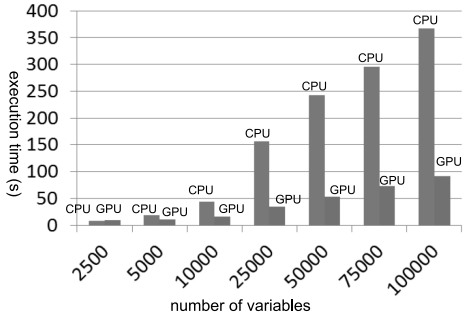


Fig. 2: A performance comparison between CPU and GPU (large instance)

results show that the proposed GPU SAT solver runs maximum 6.7 times faster than the corresponding CPU solver. One of future works is to realize more speedup by improving clause partitioning method and so on.

References

[1] Garey, M. R. and Johnson, D. S. : Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman (1979)
[2] Davis, M. and Putnum, H. : A Computing Procedure for Quantification Theory, Journal of the ACM, Vol.7, No.3, pp.201-215 (1960)
[3] Davis, M., Logemann, G., and Loveland, D. : A Machine Program for

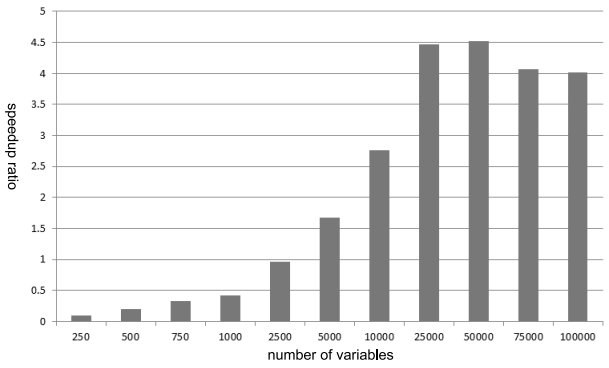


Fig. 3: Speedup ratios of GPU to CPU

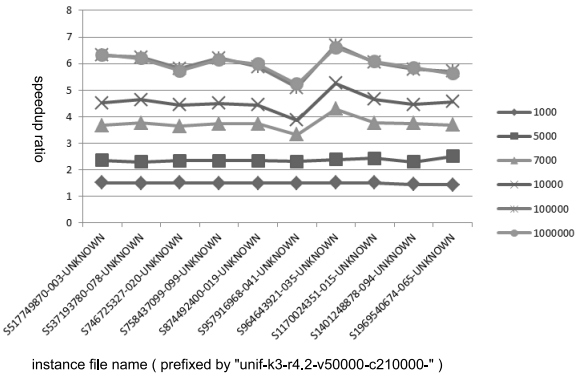


Fig. 4: Relation between speedup ratio and BCPMAX

Theorem Proving, Communications of the ACM, Vol.5, No.7, pp.394-397 (1962)
[4] Davis, J. D., Tan, Z., Yu, F., and Zhang, L. : Designing an Efficient Hardware Implication Accelerator for SAT Solving, LNCS Vol.4996, pp.48-62 (2008)
[5] SAT Research Group, Princeton University : zChaff, http://www.princeton.edu/ chaff/zchaff.html
[6] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. : Chaff: Engineering an Efficient SAT Solver, 39th Design Automation Conference (DAC), (2001)
[7] Mahajan, Y. S., Fu, Z., and Malik S. : Zchaff2004: An efficient SAT solver, International Conference on Theory and Applications of Satisfiability Testing (SAT), pp.360-375 (2004)
[8] Kirk, D. B. and Hwu, W. W.; Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann (2010)
[9] Sanders, J. and Kandrot, E.; CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional (2010)
[10] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro, Vol.28, No.2, pp.39–55 (2008)
[11] Garland, M. and Kirk, D. B.: Understanding Throughput-Oriented Architectures, Communications of the ACM, Vol.53, No.11, pp.58-66 (2010)
[12] NVIDIA: CUDA Programming Guide Version 4.0. http://www.nvidia.com/object/ cuda_develop.html (2011)
[13] NVIDIA: CUDA Best Practice Guide 4.0. http://www.nvidia.com/object/ cuda_develop.html (2011)
[14] Monien, B. and Speckenmeyer, E. : Solving satisfiability in less than 2n steps, Discrete Applied Mathematics, Vol.10, No.3, pp.287-295 (1985)
[15] Motoki, M. : SAT Instance Generation Page, http://www.is.titech.ac.jp/ watanabe/gensat/
[16] Motoki, M. and Uehara, R. : Unique Solution Instance Generation for the 3-Satisfiability (3SAT) Problem, International Conference on Theory and Applications of Satisfiability Testing (SAT), pp.293-307 (2000)
[17] Jarvisalo, M., Berre, D. L. and Roussel, O. : SAT Competition 2011, http://www.satcompetition.org/2011/ (2011)
[18] Meyer, Q., Schönfeld, F., Stamminger, M., and Wanka, R. : 3-SAT on CUDA: Towards a Massively Parallel SAT Solver, High Performance Computing and Simulation Conference (HPSC), pp.306-313 (2010)
[19] McDonald, A. and Gordon, G. : ParallelWalkSAT with Clause Learning, Data Analysis Project Presentation, School of Computer Science, Carnegie Mellon University, http://www.ml.cmu.edu/research/dap-papers/dap_mcdonald.pdf (2009)
[20] Gulati, K. and Khatri, S. P. : Boolean Satisfiability on a Graphics Processor, the 20th Great Lakes Symposium on VLSI (GLSVLSI), pp.123-126 (2010)
[21] Wang, Y. : NVIDIA CUDA Architecture-based Parallel Incomplete SAT Solver, Master Project Final Report, Faculty of Rochester Institute of Technology (2010)
[22] Deleau, H., Jaillet, C., and Krajecki, M. : GPU4SAT: Solving the SAT Problem on GPU, http://para08.idi.ntnu.no/docs/submission_49.pdf (2008)

SESSION

PERFORMANCE EVALUATION, ESTIMATION, AND RELATED ISSUES

Chair(s)

TBA

Exploring Multi-level Parallelism for Large-Scale Spiking Neural Networks

Vivek K. Pallipuram, Melissa C. Smith, Nimisha Raut, and Xiaoyu Ren
Holcombe Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634, USA

Abstract**—** *Several biologically inspired applications have been motivated by Spiking Neural Networks (SNNs) such as the Hodgkin-Huxley (HH) and Izhikevich models, owing to their high biological accuracy. The inherent massively parallel nature of the SNN simulations makes them a good fit for heterogeneous computing resources such as the General Purpose Graphical Processing Unit (GPGPU) clusters. In this research, we explore multi-level parallelism offered by heterogeneous computing resources for large-scale SNN simulations. These simulations were performed using a two-level character recognition network based on the aforementioned SNN models on NCSA's Forge GPGPU cluster. Our multi-node GPGPU implementation distributes the computations to either CPU or GPGPU based on task classification and utilizes all the available multi-level parallelism offered to ensure maximum heterogeneous resource utilization. Our multi-node GPGPU implementation scales up to 200 million neurons for the two-level network and achieves a speed-up of 355x over an equivalent MPI-only implementation.*

Keywords **–** GPGPU cluster, Neural Networks, Performance, Scalability, Multi-level Parallelism

1 Introduction

Spiking Neural Networks (SNNs) are very popular in the neuroscience community for modeling the mammalian brain to understand its functional and operational principles. The ability of spiking neurons to reproduce most of the neuronal properties with high accuracy makes them amenable for brain related studies [1]. Biologically inspired SNNs are now popular in other fields such as pattern recognition [2], artificial intelligence [3], and smart control of power grids [4]. Among several SNN models, the Izhikevich model [5] and the Hodgkin-Huxley (HH) model [6] are considered to be highly biologically accurate [1]. The Izhikevich model is the most recent and computation efficient model, whereas the HH model is the oldest and highly computation intensive model.

With the advent of General Purpose Graphical Processing Units (GPGPUs) in the field of high performance computing, the current trend is to extract concurrency from heterogeneous computing resources such as GPGPU clusters. The current state-of-the-art heterogeneous systems are composed of several thousands of compute nodes, where each node is equipped with multiple CPU cores in conjunction with one or more GPGPU accelerators. Since GPGPUs have been established in the literature as viable architecture choice for SNN simulations [7, 8], GPGPU clusters are lucrative options for large-scale SNN simulations and related studies. Although these heterogeneous systems can provide substantial performance for massively parallel simulations, much of their computing resources are often under-utilized due to poor tuning strategies. To achieve optimal utilization of the heterogeneous resources, it is imperative to perform efficient load-balancing between the CPU cores and GPGPU devices, which requires exposing all available parallelism in the application and effective memory and bandwidth utilization.

With the above as motivation, in this research, we investigate multi-level parallelism for large-scale massively parallel SNN simulations. These simulations were performed using a two-level character recognition network based on [2]. The two-level network capable of recognizing 48 alpha-numeric characters was developed using the Izhikevich and HH models. The two-level network was mapped on the *National Center for Super-Computing Applications* (NCSA) 32-node Forge GPGPU cluster and scaled to over 200 million neurons. The focal contributions of this work are:

1) Exploration of multi-level parallelism for spiking neural networks in the form of GPGPU+MPI+OpenMP implementations.
2) Demonstration of optimal load-balancing between the CPU cores and GPGPUs.
3) Scaling of the two-level network beyond the single-node capability.
4) Scalability and performance analysis on the Forge cluster.

Our heterogeneous parallel implementations were able to achieve significant application speed-ups, as high as 355x for the computation intensive HH model and 4x for the computation efficient Izhikevich model over equivalent MPI-only implementations on the Forge cluster. The performance of the two SNN models was found to depend on the computation-to-communication requirements of the SNN models.

The rest of the paper is organized as follows. Section 2 provides a brief background on the two SNN models, the two-level network, and an overview of the GPGPU architecture. Section 3 discusses related work. Section 4 details the experimental set-up and network mapping. Section 5 presents the results and analysis, and the paper is concluded in Section 6 with conclusions and suggestions for future work.

2 Background

2.1 Spiking neural networks

Over the last 50 years, several models [9] have been proposed that capture the spiking mechanism within a neuron. In this paper, we examine two of the most biologically accurate spiking neuron models for implementation on the GPGPU cluster. In what follows, we give a brief chronological overview of these two popular SNN models, namely the Hodgkin-Huxley and Izhikevich models.

The Hodgkin–Huxley (HH) model is considered to be the most accurate and the most important model in the neuroscience community till date. As mentioned in [1], the model involves 4 equations and 10 parameters describing various neuron current activation and deactivation. The model takes 120 flops per 0.1 ms time-step and hence 1200 flops/1 ms for the update. In our research, we have used 0.01 ms time-step for the neuron update.

In [5], Izhikevich developed a simple and very computation efficient spiking neuron model that has similar accuracy as the HH model. Izhikevich successfully reduced the complex HH model equations to a 2-D system of ordinary equations. Izhikevich's model requires only 13 flops per neuron update and still sufficiently reproduces a majority of neuronal properties. The equations are found in [5]. In our research, we have used a 1 ms time-step (13 flops/1 ms update) for neuronal dynamics update for the Izhikevich model.

The time-steps used in our research for the models discussed are in the valid range of time-steps that are deemed sufficient for reproducing biologically relevant neuron dynamics [1].

Table I provides the Flops/Byte ratio for the two models, which is pertinent to the performance analysis of the two models. The Flops/Byte ratio is an algorithm specific value and is defined as the ratio of the number of floating-point operations required for a complete neuron update (level-1 and level-2) to the overall bytes requested (all model parameters, firing vector and block firing vector) for all of the neuron updates [8].

Table I. Flops/Byte ratio for the two models

Model	Flops/neuron required for the complete neuron update	Flops/Byte ratio
HH	246	9.84
Izhikevich	13	0.9997

2.2 The two-level network

The two-level character recognition network used in this research is based on [2] and the network used to test the models is shown in Figure 1. The task of the network is to detect images from a training data set of 48 images. The level-1 neurons act as an input collection layer and the level-2 neurons act as output collection layer. Each neuron in level-1 corresponds to a pixel in the input image; hence the number of neurons in the input level is equal to the total number of pixels in the test image. The number of neurons in the output layer, level-2, is equal to the number of images in the database. When an input image is presented to level-1, each neuron evaluates its membrane potential based on the pixel level presented and the neuron model chosen. This process is referred to as the *evaluation of neuron dynamics*. If the pixel is "on," a constant current is supplied to the neuron for membrane potential evaluation. The input current equation for a level-2 neuron is:

$$I_j = \Sigma w(i,j) f(i) \qquad (1)$$

In (1), I_j is the net input current to the neuron j in level-2, $w(i,j)$ is the weight of the synapse connecting neuron i in level-1 and the neuron j in level-2. A neuron in any level is said to have "fired" if its membrane potential crosses the threshold value for the selected neuron model. The research presented in this paper accelerates the recognition phase of the network by implementing all of the level-1 neurons on the GPGPU devices, while the level-2 neurons (input current accumulation and dynamics) are implemented on the host processors as will be discussed later in Section 4.

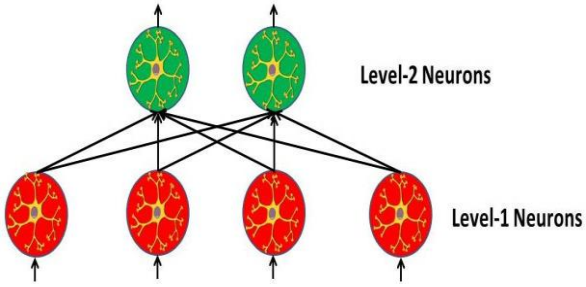


Figure 1. Two-level character recognition network

2.3 The GPGPU architecture

The Compute Unified Device Architecture (CUDA) programming framework views the GPGPU architecture as an array of streaming multi-processors (SMPs). Each multi-processor contains a set of scalar processors (referred to as CUDA cores), a double-precision (DP) unit, shared memory for thread cooperation, and texture addressing and texture fetch units. While a single thread is executed on a CUDA core, a group of threads called a *thread block*, is executed on the SMPs. Threads in a thread block can synchronize with each other using shared memory. The Fermi architecture has brought a lot of innovation versus previous Nvidia architectures: 512 CUDA cores organized as 16 SMPs with 32 cores each sharing a L2 cache. The SMPs have two sets of 16 CUDA cores, 4 special function units for transcendental functions, 16 load/store units, a hefty register file, and a configurable 64 KB L1 cache/shared memory. The GPGPU device has the capability of supporting 6 GB of GDDR5 DRAM memory. The Fermi-based Tesla M2070

used in this research can theoretically offer 1.03 tera-flops of single-precision floating-point performance and 515 giga-flops of double-precision floating-point performance. More information on the GPGPU architecture and CUDA framework can be found in [10] and [11], respectively.

3 Related work

Several research activities have been motivated by the idea of modeling the neocortex. In [12], the authors have studied the mammalian brain neocortex in detail and were successful in simulating a rat-size cortex in 42% of real-time and a cat-size cortex in 23% of real-time on a 442 node Dell Xeon cluster. Their neuron model is in the integrate-and-fire (I&F) category, which according to Izhikevich, is insufficient for accurately reproducing neuronal properties [1]. In [13], the authors successfully used Izhikevich's model to simulate a cat-size cortical model with 10^9 neurons and 10^{13} synapses using a BlueGene/P machine with 147,456 processors and 144 TB of main memory. The authors claim that their simulation scale is roughly 1–2 orders of magnitude smaller than the human cortex and 2–3 orders of magnitude slower than real-time.

Alternative computing architectures such as GPGPUs are now being investigated for biologically realistic simulations. In [7], the authors implemented Izhikevich's random network on Nvidia's GTX-280 with 1 GB memory and achieved a speed-up of 26x over an equivalent software implementation for a 100K neuron network simulation. Their work discusses mapping strategies on the GPGPU to efficiently utilize the memory bandwidth and parallelism.

In [14], the authors investigated GPGPU cluster based implementations of the HH and Izhikevich models using a two-level network based on [2]. They reported GPGPU speed-ups of 24.6x and 177x for the Izhikevich and HH models, respectively over a 2.4 GHz dual-core AMD opteron processor. Their 16 GPGPU-based MPI implementation on a 32-node Tesla S1070 NCSA cluster was successful in scaling the network up to 150 million neurons and achieved 17910 ms runtime for the HH model.

Although our two-level network and experiments are similar to [14], our work is different in the following ways. In [14], the authors perform the level-1 neuron dynamics, level-2 current accumulation, and level-2 neuron dynamics calculations on the GPGPU device. In [14], the authors also claim that the image detection operation (checking if any of the level-2 neurons fired) is inherently serial, therefore it can be performed on the CPU host. As will be highlighted in Section 4, the implementation in [14] is not efficient since the level-2 dynamics calculation is fairly small (only 48 neurons in level-2), performing this operation on the GPGPU device is not warranted and renders the CPU resources under-utilized. Additionally, the evaluation of level-2 neuron currents on the GPGPU device requires the transfer of the large weight matrix to the GPGPU device memory, which is wasteful of the host-device bandwidth and device memory as explained in [15]. Our GPGPU-based MPI implementation achieves optimal load-balancing between the CPU cores and GPGPU devices by implementing the level-1 neuron dynamics on the GPGPU devices and level-2 neuron calculations (current accumulation and dynamics) on the host processors as will be discussed in Section 4. In addition, our work exploits parallelism across multiple levels of the heterogeneous architecture in the form of a complete GPGPU+MPI+OpenMP based implementation.

4 Experimental setup and network mapping

4.1 Experimental setup

NCSA's Forge GPGPU cluster was used in this research for the large-scale SNN simulations. The 153 tera-flop cluster is composed of 36 Dell PowerEdge C6145 servers; each server is connected to six Fermi-based Tesla M2070 GPGPUs via three PCI-e Gen2 X16 slots. Each server is equipped with two 2.4 GHz AMD Opteron Magny-Cours 6136 processors, eight cores each. The network interconnect is comprised of InfiniBand QDR. Our implementations were developed using CUDA 4.0 and OpenMPI 1.4.3 on Red Hat Enterprise Linux 6.

4.2 SNN network mapping

In this sub-section, we first provide the details of network mapping for the single-GPGPU implementation that is subsequently extended to a GPGPU-based MPI implementation.

As discussed in section 2.2, level-1 is the most computation intensive layer of the network since the number of neurons is equal to the number of pixels in the input image; therefore these operations are performed on the GPGPU device. Each GPGPU thread evaluates the dynamics of a single level-1 neuron. Therefore, the number of GPGPU threads created is equal to the number of level-1 neurons. The GPGPU device then provides the host processor with the level-1 neuron firing information, the *global firing vector*, which is used by the host processor to obtain the level-2 neuron currents and dynamics. The level-2 computations (current accumulation and dynamics) are implemented on the host processor since the level-2 neuron computations constitute less than 5% of the total computation overhead and, implementing the level-2 dynamics on the GPGPU would require the transfer of the weight matrix to the GPGPU device memory. Hence any computational improvement obtained by implementing level-2 neuron dynamics will be insufficient to amortize the communication overhead involved in transferring the large weight matrix to the GPGPU device. The single-GPGPU implementation was optimized with memory level, instruction level, and execution configuration level optimizations as mentioned in [15].

The host-device bandwidth was further optimized using a *block firing vector* concept introduced in [8]. The block firing vector is implemented in the device shared memory to avoid transferring the global firing vector in each algorithmic time-step. The block firing vector is similar to the global firing vector but instead acts as a collection of flags for thread blocks. Since the threads are collected in thread blocks of size: *blocksize*, the block firing vector is *blocksize* magnitude smaller than the global firing vector, and hence can be transferred from the device to host in each time-step with minimal overhead. If at any time-step the block firing vector contains information of a firing event, only then will the entire global firing vector be transferred from the device to host and then read by the host. Figure 2 illustrates the block firing vector concept.

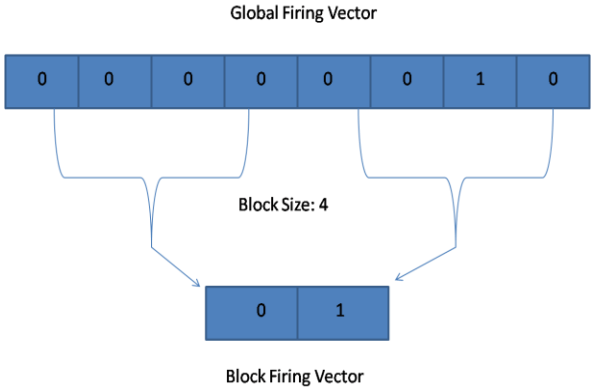


Figure 2. The concept of block firing vector

The single-GPGPU implementation is then extended to a GPGPU-based MPI implementation. The MPI ranks were assigned in node-packing fashion, meaning the ranks are packed into nodes. The nodes were configured with a maximum of six MPI processes per node. This configuration allows for 1:1 CPU core-to-GPGPU ratio at each node and potentially reduces long distance inter-node communication. The GPGPU devices were allotted to the CPU cores using modulo rule where an MPI process with rank *n* is coupled with the GPGPU device number, *n modulo 6* [16]. Future work will investigate the impact of other CPU core-to-GPGPU ratios on application performance.

The GPGPU-based MPI orchestration is as follows. The MPI rank 0 acts as the master process that scatters the level-1 neuron inputs to all other processes. The level-1 neuron parameters are initialized to the SNN model specific constant values at each MPI process, and hence require no MPI communication. Each CPU-GPGPU pair works as an independent unit where the GPGPU device evaluates the partial level-1 neuron dynamics and the host processor evaluates the partial level-2 currents using the firing vector obtained from its designated GPGPU device. The partial level-2 currents from each MPI process are then accumulated at MPI rank 0 where the complete level-2 neuron dynamics are evaluated and the image detection decision is made. The level-2 neuron computations on the hosts were accelerated using OpenMP. Figure 3 elucidates the orchestration of the GPGPU-based MPI implementation discussed in this sub-section.

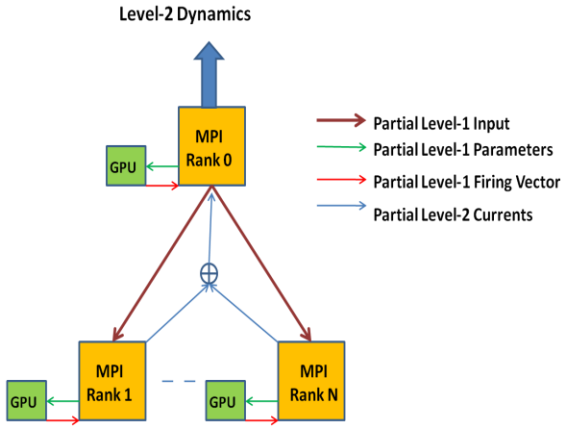


Figure 3. GPGPU-based MPI orchestration

5 Results and analysis

In this section, we present the results for the GPGPU-based MPI implementation of the two-level network developed using the HH and Izhikevich SNN models. We discuss the application runtime values, show the overall runtime breakdown in terms of GPGPU time, CPU time, and communication time for a 32 CPU Core-GPGPU device pair cluster configuration, and compare the GPGPU-based MPI implementation with an equivalent MPI-only implementation. A single CPU Core-GPGPU device pair shall henceforth be referred to as a host-device pair. For the two SNN models, the cluster configuration was varied from 2 up to 32 host-device pairs. We first present the results for the computation intensive HH model followed by the results for the Izhikevich model.

5.1 Hodgkin-Huxley model

The statistical-average runtime values for different cluster configurations versus the network size using the HH model are given in Table II. These runtimes correspond to those measured by the master process, MPI rank 0, which distributes the tasks and makes the final image detection decision. The implementation successfully scaled the two-level network to 200 million neurons using a configuration of 32 host-device pairs with a statistical-average runtime of 2416 milliseconds. The dashes in the table indicate that the problem will not fit in the GPGPU device memory resulting in a configuration failure for that particular neural network size.

Table II. HH model: Statistical-average runtime values (in ms)

Cluster Configuration	Network Size (in millions)			
	9.73	51.8	92.16	207.36
2	1256	-	-	-
4	742	3516	-	-
8	486	1928	3396	-
16	412	1168	1990	-
32	356	784	1195	2416

As seen in Table II, the scalability of the implementation improves with the increase in network size. We define the runtime *improvement ratio* as the ratio of runtimes of two successive cluster configurations for a given network size. For a network size of 9.73 million neurons, the runtime improvement ratio is 1.7 for 2 vs. 4 host-device pairs, 1.5 for 4 vs. 8 host-device pairs, 1.2 for 8 vs. 16 host-device pairs, and 1.15 for 16 vs. 32 host-device pairs. However, for a large neural network size, 51.8 million neurons, the improvement ratios are better with values 1.8, 1.65, and 1.5 for 4 vs. 8, 8 vs. 16, and 16 vs. 32 host-device pairs, respectively. The above scaling behavior is expected since the amount of computations per GPGPU device decreases with the host-device pair scaling. Consequently, for smaller network sizes, the GPGPU computations are not sufficient to amortize the necessary CPU computations and MPI communications.

Figure 4 further supports the scalability explanation given above. The figure provides the overall runtime broken into: GPGPU time (kernel time and host-device transfer time), CPU time (level-2 currents and dynamics), and MPI communication time for a 32 host-device pair configuration versus the network size. For a small network configuration of 13 million neurons, CPU time dominates the GPGPU time owing to a relatively small number of computations per GPGPU device. As the network size increases, the number of computations per GPGPU device increases significantly, thereby making the GPGPU time dominant with respect to the overall runtime.

Figure 5 provides the speed-up of the GPGPU-based MPI implementation over an equivalent MPI-only implementation. The 32 host-device pair configuration was able to achieve a speed-up of 281.8x over the equivalent MPI-only implementation for the largest SNN network size. Table III provides the speed-up values for many of the intermediate network sizes tested. As shown in Figure 5 and Table III, the speed-up over the equivalent-MPI implementation increases with the increase in network size for all of the cluster configurations. The increase in speed-up is due to the amortization of CPU and MPI communication times by the GPGPU computations due to the increased number of GPGPU computations required by the increasing network size. The speed-up values are particularly large for the HH model due to its high Flops/Byte ratio requirements (see Table I). This supports the claim that applications with high Flops/Byte ratios are highly suited for GPGPU-based implementations [8].

Further inspection of Figure 5 and Table III reveals that for a fixed network size, the speed-up of the GPGPU-based MPI implementation over the equivalent MPI-only implementation falls as the number of processors increases. As explained previously, a significant number of computations are required to fully utilize the compute capabilities of the GPGPU device; hence large cluster configurations observe lower speed-ups for smaller network sizes.

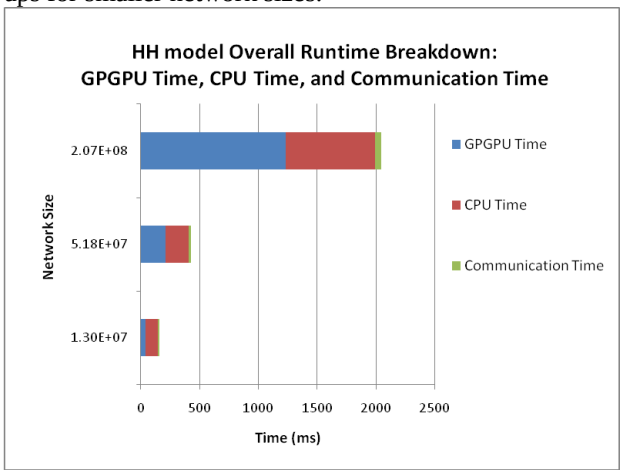


Figure 4. HH model: Overall runtime breakdown for 32 host-device pair configuration

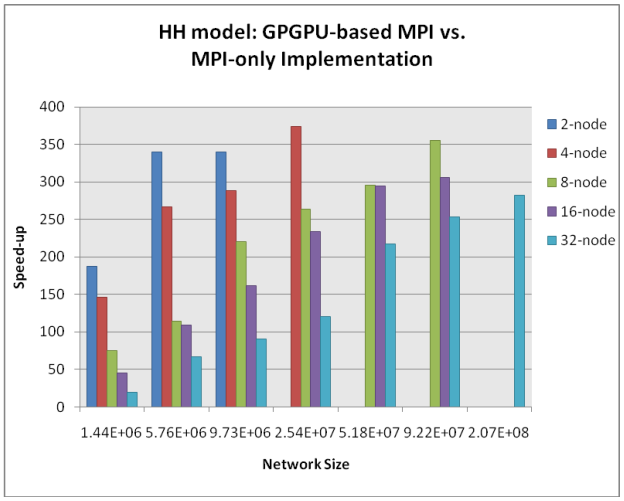


Figure 5. HH model: GPGPU-based MPI vs. MPI-only implementation

Table III. HH model: GPGPU-based MPI vs. MPI-only implementation

Cluster Configuration	Network Size (in millions)			
	1.44	**9.73**	**25.4**	**92.2**
2	187x	340x	-	-
4	146x	288x	374x	-
8	75x	220x	264x	355x
16	44x	162x	233x	306x
32	20x	90x	120x	253x

5.2 Izhikevich model

The statistical-average runtime values for different cluster configurations versus the network size using the Izhikevich model are given in Table IV.

Table IV. Izhikevich model: Statistical-average runtime values (in ms)

Cluster Configuration	Neural Network Size (in millions)			
	9.73	**51.8**	**921.16**	**207.36**
2	180	-	-	-
4	142	928	-	-
8	118	952	725	-
16	96	363	453	938
32	96	491	253	498

Unlike the high Flops/Byte ratio HH model, strong scaling is not observed for the low Flops/Byte ratio Izhikevich model as seen in Table IV. In addition to the lower number of computations in the Izhikevich model (see Table I); the fall in the number of computations per GPGPU device further impedes the scaling performance. Figure 6 provides the overall runtime breakdown for the 32 host-device pair configuration in terms of CPU time, GPGPU time, and communication time.

As seen in Figure 6, the CPU time continues to dominate the kernel time as the network size increases, leading to sub-optimal performance for the Izhikevich model. The continued domination of the CPU time is due to the increased level-2 current computations as the network size increases. Although computations per GPGPU device also increase with the increase in network size, the increase is marginal due to nominal number of computations in the Izhikevich model.

Figure 7 and Table V present the performance comparison of the GPGPU-based MPI implementation and MPI-only implementation. The 32 host-device pair configuration attained a speed-up of 2.87x versus the 32-processor MPI-only implementation. As seen in Table V, the increase in speed-up with the increase in network size is marginal for all cluster configurations examined. The explanation for the fall in the speed-up with the increase in cluster configuration for fixed network size is the same as was given for the HH model.

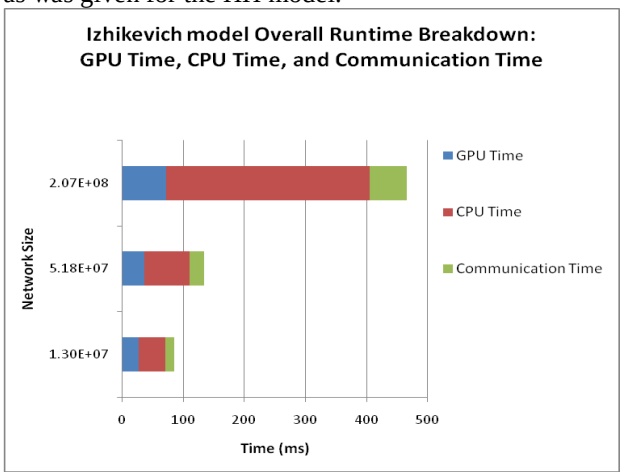


Figure 6. Izhikevich model: Overall runtime breakdown for 32 host-device pair configuration

The Izhikevich model is an interesting case for GPGPU-based MPI implementation. Although the application itself is massively-parallel, it involves only a nominal amount of computations. Therefore, the GPGPU computations do not amortize the increased CPU computation and MPI communication overhead as the SNN network size increases. As mentioned in this sub-section, the level-2 current evaluation for the Izhikevich model increases significantly with the SNN network size. One possible improvement is to implement the level-2 current evaluation on the GPGPU device for larger network sizes. The task would not only require meticulous handling of the GPGPU device memory for the inherent reduction operation involved, but also accommodation of the large weight matrix (48 * Network Size) in the GPGPU device memory. The Izhikevich model explored in this research serves well to highlight the importance of an optimal application-to-accelerator cluster match. It is claimed that applications should not only expose sufficient parallelism, but should also yield enough computations to fully utilize the compute capabilities of heterogeneous clusters. Nonetheless, our GPGPU-based MPI implementations produced performance advantages versus the equivalent MPI-only implementations as shown in this section.

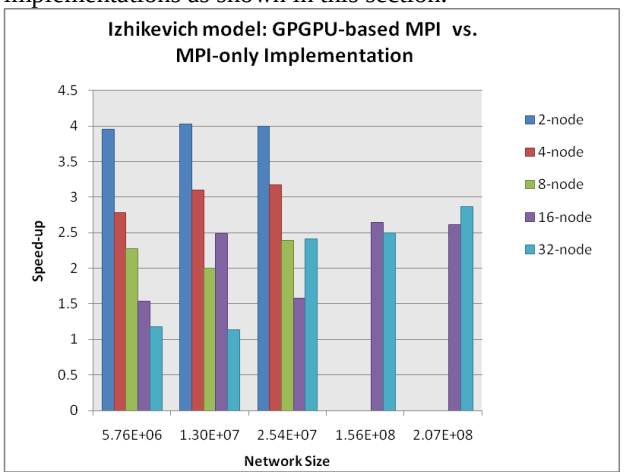


Figure 7. Izhikevich model: GPGPU-based MPI vs. MPI-only implementation

Table V. Izhikevich model: GPGPU-based MPI vs. MPI-only implementation

Cluster Configuration	Neural Network Size (in millions)			
	5.76	13	25.4	156
2	3.9x	4.0x	4.0x	-
4	2.8x	3.0x	3.2x	-
8	2.3x	2.0x	2.4x	-
16	1.5x	2.5x	1.6x	2.7x
32	1.2x	1.1x	2.4x	2.5x

6 Conclusions and future work

In this research, we explored the multi-level parallelism offered by heterogeneous GPGPU clusters for large-scale SNN simulations in the form of a complete GPU+MPI+OpenMP implementation. A GPGPU-based MPI orchestration was presented that allows for optimal heterogeneous resource utilization for large-scale SNN simulations. The two-level network based on the HH and Izhikevich SNN models successfully scaled to 200 million neurons using a 32 host-device pair cluster configuration. In addition to providing significant speed-ups, as high as 355x over an equivalent MPI-only implementation, the GPGPU-based MPI implementation for the HH model scaled well with the SNN network size. Although, our GPGPU-based MPI implementation for the Izhikevich model performed slightly better than the MPI-only implementation, sub-optimal scaling was observed with increasing SNN network size. The results for the Izhikevich model implementation highlighted the

importance of an optimal application-to-accelerator cluster match for maximum application performance. It is claimed that applications should not only expose sufficient parallelism, but should also yield enough computations to fully utilize the compute capabilities of the heterogeneous cluster resources.

As future work, we plan to explore other cluster configurations with different CPU Core-to-GPGPU device ratios per node and investigate the performance of such configurations for large-scale SNN simulations. The future work also includes the development of performance prediction models for heterogeneous clusters that optimally matches the applications with appropriate heterogeneous cluster configurations. As encouraged by the large-scale simulation efforts presented in this research, we also plan to explore GPGPU cluster based smart control of power grids that employs biologically inspired SNNs as massively parallel computation engines.

7 Acknowledgement

This work was supported in part by the National Science Foundation under Grant No. CCF-0916387. The authors gratefully acknowledge the National Center for Super-Computing Applications (NCSA) for granting access to their computing resources.

8 References

[1] E. Izhikevich, "Which Model to Use for Cortical Spiking Neurons?", *IEEE Transactions on Neural Networks*, vol. 15(5), pp. 1063-1070, 2004

[2] A. Gupta, L. Long, "Character Recognition using Spiking Neural Networks," in *Proc. IJCNN*, pp. 53 – 58, August 2007

[3] D. Surdilovic, J. Radojicic, M. Schulze, M. Dembek, "Modular hybrid robots with actuators and joint stiffness control", in *Proc. BioRob*, pp. 289-294, October 2008

[4] C.E. Johnson, "Spiking Neural Networks and their Applications", Ph.D dissertation, 2011

[5] E. M. Izhikevich, "Simple Model to Use for Cortical Spiking Neurons," *IEEE transactions on Neural Networks*, vol. 14, no. 6, pp. 1569-1572, November 2003

[6] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and application to conduction and excitation in nerve," *Journal of Physiology*, vol. 117, pp. 500-544, 1952

[7] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbauma, "A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors," *Special issue of Neural Network, Elsevier*, vol. 22(5-6), pp. 791-800, July 2009

[8] V. K. Pallipuram, "Acceleration of Spiking Neural Networks on Single-GPU and Multi-GPU systems", Master's Thesis, 2010

[9] E. M. Izhikevich, "Dynamical Systems in Neuroscience," *MIT press*, Cambridge, Massachusetts, 2007

[10] "NVIDIA's Next Generation CUDA Compute Architecture: Fermi", http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf

[11] "NVIDIA CUDA Programming Guide", http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

[12] C. Johansson and A. Lansner, "Towards Cortex Sized Artificial Neural Systems," *Neural Networks*, 20(1), pp. 48-61, January 2007

[13] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha, "The Cat is Out of the Bag: Cortical Simulations with 109 Neurons, 1013 Synapses," *Proceedings of SC '09*, Portland, Oregon, November 2009

[14] B. Han, T. M. Taha, "Neuromorphic models on GPGPU cluster", in *Proc. IJCNN*, pp. 1-8, July 2010

[15] V. K. Pallipuram, M. A. Bhuiyan, M. C. Smith, "A Comparative Study of GPGPU Programming Models and Architectures Using Neural Networks", *Journal of SuperComputing*, Springer Publications, DOI 10.1007/s11227-011-0631-3

[16] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, W. Hwu, "GPU Clusters for High-Performance Computing", *PPAC 2009*, in *Proc. IEEE Cluster 2009*, August 2009

A Methodology for generating Dynamic Tuning strategies in Multicore Systems

César Allande[1], Josep Jorba[2], Anna Sikora[1] and Eduardo César[1]
[1]Computer Architecture and Operating Systems Department (CAOS)
Universitat Autònoma de Barcelona, Barcelona, SPAIN
[2]Universitat Oberta de Catalunya (UOC), Barcelona, SPAIN

Abstract— *One of the main consequences of current HPC systems heterogeneity is that different levels of parallelism should be considered in all phases of parallel application development. Therefore, support tools and, in particular, performance analysis and tuning tools, must also be adapted to manage heterogeneity. A significant step forward in this adaptation consists of developing specific strategies to automatically improve the performance of the parallel regions of the application being executed in each computing element. We propose a methodology to systematically develop performance optimization strategies for specific application patterns taking into consideration hardware characteristics. These performance optimizations are intended to be applied by means of the management code provided by most high level libraries. This study describes the methodology developed and shows how it can be used to expose performance factors that can be dynamically tuned on an OpenMP application.*

Keywords: OpenMP, performance analysis, dynamic tuning

1. Introduction

Performance on multicore systems does not depend only on the processor frequency, but mainly on the number of cores, so, intuitively more cores means higher performance. However, cores have shared hardware resources and performance also depends on how concurrency and contention are managed, making it more difficult to meet performance goals.

Nowadays hardware architectures have such heterogeneity that programming frameworks/languages have to take into consideration many possible configurations, going from multicore processors based on a shared memory hierarchy, to manycore proposals based on cores interconnected with mesh on-chip network (Tilera[1]) and dedicated hardware accelerators (GPUs). Nowadays, processor architecture designs tend towards an increasingly higher number of cores in the same processor. The way parallel programming frameworks (defined in a comprehensive way, i.e., including libraries, languages and so on) are going to manage those cores involves a high complexity.

There is a widerange of programming frameworks for multi/many core architectures depending on the implemented programming model. In addition, other models like MPI, have implementations that take advantage of the shared memory hierarchy for communicating processes in the same node. However, shared memory architecture of most multicore systems is usually managed by frameworks implementing the shared memory programming model, such as OpenMP [2], Cilk [3], and so on. Consequently, developing parallel applications is becoming more and more difficult because of heterogeneous architectures and corresponding heterogeneous programming frameworks. Logically, this complexity is also reflected in the performance analysis and tuning process of these applications.

Therefore, due to the growing heterogeneity of multicore systems and the configurability of the parallel programming frameworks, a way for improving the applications' performance is to dynamically adjust the applications' requirements to the characteristics of each hardware architecture using the code provided by the parallel programming framework.

OpenMP is one of the most widely spread APIs for multiplatform shared-memory parallel programming in C/C++ and Fortran. The management of parallelism is made through a library that can be adapted for deploying performance strategies. To adjust the runtime execution, it is possible to use dynamic instrumentation tools, such as Pin[4] or Dyninst[5].

We propose a methodology, that considering an application, an architecture, and a runtime manager, would determine the implication of performance factors, such as loop management, data imbalance, thread imbalance, and so on; and that will help to provide dynamic strategies that can be applied to those factors.

Dynamic tuning is a complex undertaking, particularly since many performance factors and mitigation strategies are not independent. This paper is focused in discussing a methodology for developing these strategies, the defining elements that have been considered, and its objectives. In addition, a complete example of its use is included for a particular application.

To achieve these objective, this works has been structured as follows. Related work is presented in Section 2. Section 3 describes the methodology. This is followed by the methodology applied to a case of study in Section 4 and in Section

5 is described the dynamic tuning tool developed for the performance factor exposed on the case of study. Finally, conclusions are detailed in Section 6.

2. Related work

Dynamic instrumentation is used in performance analysis environments to tune applications at runtime. As an example, MATE[6] performs automatic dynamic tuning by inserting code into the application through the Dyninst library. This framework uses externally provided strategies for taking tuning decision, but it is specific for distributed memory systems and there is no implementation for multicore.

Wicaksono et al.[7] demonstrates the functionality of a collector-based dynamic optimization framework called DARWIN that uses collected performance data as feedback to affect the behavior of the program through the OpenMP runtime. It is able to take different actions, such as modifying the number of threads, adjusting core frequency, or accessing a specialized malloc library transforming the source code, or modifying instructions in the binary. This framework has been evaluated in a ccNUMA platform, therefore, the main performance problems are related to scalability due to data locality between nodes and false sharing detection. However, this global approach does not consider hybrid systems (MPI+OpenMP) where the OpenMP problems are local to the node. Our methodology is oriented to nodes, by considering that, nowadays, most scientific applications use hybrid programming models and improving a local execution will benefit the overall performance.

Other works such as Stephen Olivier et al.[8] and A.Duran et al.[9], are focused on performance issues related to task managing. The first discusses task managing work-stealing algorithm, where it is necessary to define the appropriate number of tasks to be stolen from a remote queue, as well as the performance results achieved on several architectures. While the second proposes a cut-off strategy for limiting the space required to deploy tasks. Various strategies are evaluated to reduce the number of created tasks, and consequently the creation overhead is reduced.

In addition, the utilization of different compilers could be determinant for tasks based applications performance, as shown in an evaluation for different compilers supporting task parallelization is shown in Stephen Olivier and Jan Prins[10].

Up to date, most scientific applications in OpenMP are data parallel. Consequently, we consider that dynamic performance tuning has to take into consideration both, performance related to task and data parallelism, and we have developed our methodology in accordance to this consideration.

3. Methodology

The objective of the proposed methodology is to define models and tuning strategies associated to performance factors in multicore systems. A performance factor is a latent performance problem that can arise in a specific context. The performance factors depend on the application patterns, the high level library, and hardware characteristics.

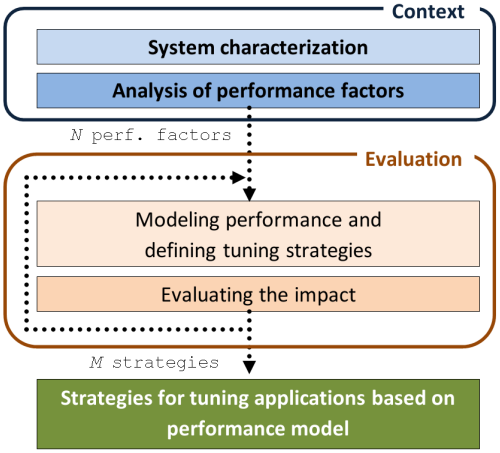


Fig. 1: Methodology for generating dynamic tuning strategies in multicore systems

The methodology has to provide, for each performance factor: what should be measured, the performance model, and the strategies for automatically tuning the application. Fig. 1 shows that the context of the methodology is defined on the first two stages. The first is a characterization of the application, the manager library, and the hardware. Then, this characterization has to provide insight on related performance factors that are affecting the application's performance. This is done generating hypotheses of performance factors.

Afterwards, the next stages are developed for an evaluation on a specific context, trying to achieve the most general solution for a performance factor. Therefore, to find a valid performance model strategy for a performance factor, the methodology has to iterate in order to evaluate different context configurations to validate the proposed strategy.

The general context is presented in the next subsections and summarized in Fig. 2.

3.1 System characterization

As mentioned before, it is necessary to take into consideration the application pattern, and the manager library and hardware specific characteristics.

Latent performance factors may arise when context changes. To observe context implications, it is necessary to analyze static and dynamic information at different levels.

Static information has to be provided from context specification, such as applications configuration (data input parameters, pattern, parallel library), and systems libraries (parallelization paradigm, scheduling parameters), and architecture specification (number of cores, memory hierarchy, and so on).

Dynamic information can be analyzed through micro benchmarking. This information provide context's behavior and some of its low level characteristic, such as system's GFlops on computation bound performance analysis, evaluation of performance for different affinity configurations, and so on.

3.1.1 Application

Scientific programs can be classified by its pattern design (Master/Worker, Pipeline, SPMD), but at thread level those patterns could not present specific performance factors. Moreover, there are more appropriated characteristics in threaded applications such as iterativity, recursivity, computation bound, memory bound, and so on. These properties can better represent the characteristics and related performance factors of a threaded application.

Performance factors at application level are usually coarse-grained, and performance strategies can hardly be generalized. Thus, the detection of these patterns may allow the use of dedicated strategies. For example, some iterative patterns generate thread imbalance by working with umbalanced data. It is possible, to estimate data input patterns, on the first iterations of an application, in order to try to balance the rest of the execution. However, this is a coarse grained strategy heavily linked to the pattern and hardly generalizable.

As an experimental testbed, we use scientific kernels present in different OpenMP benchmarks as NAS Parallel Benchmark[11] and BOTS[12] (Barcelona OpenMP Tasks Suite) that are representative of some communication and computing patterns observed in scientific multicore applications.

3.1.2 Manager library

One of the most widely used programming model implementation is OpenMP, as a superset of threads library. In its last specification it allows data and task parallelism. Thus, the performance factors can be splitted into the specific context of the data parallelism common performance factors related to loop directives, affinity definition, synchronization elements and parallel constructs; and in the case of task parallelism the performance factors are related to scheduling policies (centralized or decentralized), task granularity, number of tasks, and creation overheads.

Some of those performance factors can be measured with specific benchmarks such as the EPCC[13] micro benchmark suite, that allows for studying of OpenMP performance comparisons between different OpenMP implementations.

3.1.3 Hardware

It is important to have an initial knowledge of our hardware system configuration. The hardware performance factors are mainly related to computing capabilities and memory architecture.

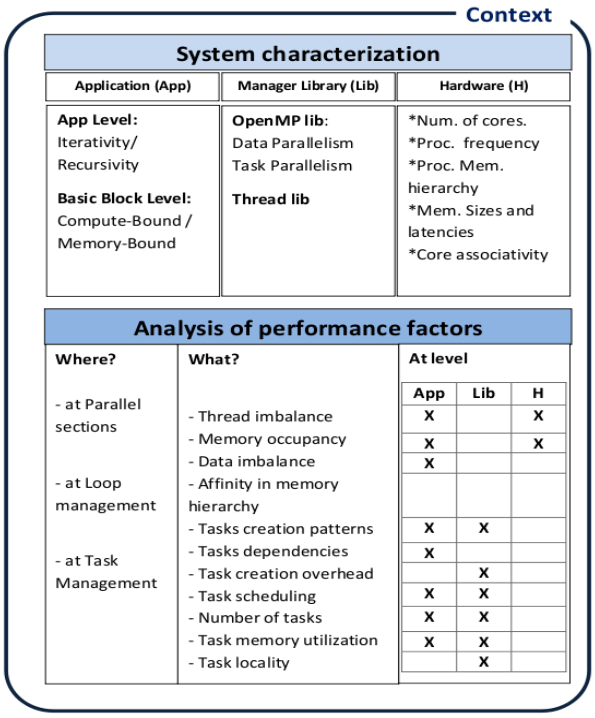


Fig. 2: Context characteristics development template

To analyze the hardware characteristics it is possible to evaluate the dynamic behavior by using benchmark tools. The computing capabilities can be measured with a compute-bound application that can provide metrics such as GFlops. Also, measuring memory issues through benchmarking can provide an insight of the memory architecture. Some benchmark application use threads, therefore, it can be significant to evaluate the system for different numbers of threads and different affinity configurations.

3.2 Analysis of performance factors

The analysis of an application has to provide the inherent problems that can arise at execution time. In order to find the dynamic application problems it must be evaluated for different contexts. A problem can arise, or its impact can affect performance, depending on the architecture, the runtime manager, and the application configuration. When a performance factor is detected it has to be isolated, as much as possible, to identify its sources.

A first step is to identify where the latent performance factor is located, generally associated to parallelized code. The performance issues in OpenMP applications are present on parallel sections, loop regions, and task regions. Each of which are managed by specialized functions of the dynamic library.

The analysis of performance factors, defined into a context, provides a knowledge base. Presumably, the strategies applied for an application could be applied for other similar applications in the same context.

The following performance factors have been classified to be considered by a dynamic tuning tool, in order to feasibly apply dynamic tuning strategies.

1) Thread imbalance; can arise at application level due to data dependencies, synchronization points, and also because of different latencies on core memory access. It is possible to dynamically adjust the number of computing elements and vary the affinity definition, or the scheduling policy, in order to minimize the imbalance.
2) Memory occupancy; threads accessing memory can cause memory contention for the increase of replacements at shared cache level, generating performance degradation. A bad definition at application level can origin this performance factor. The performance degradation can be decreased dynamically by adjusting the number of racing threads.
3) Data imbalance; differences in the amount of data assigned to each thread can cause imbalance. This performance factor is implicit to the application. However, adequate dynamic strategies can be applied for data partitioning and allocation, or data reordering, that have demonstrated to improve performance in some cases.
4) Affinity in memory hierarchy; when an application is executed on an architecture, a mismatch between the application configuration and computing elements sharing memory resources can originate performance degradation. Trying to take advantage of collaborating threads and memory re-utilization sometimes can cause a false-sharing condition. Analyzing the behavior of the application and assigning collaborative threads to the appropriate hierarchy levels will promote collaborative computation.
5) Task management; there are an extense variety of performance factors in the task parallel model. Tasks, as executing units, are also affected by the same performance factors that in the data parallel model. Furthermore, due to its highly dynamic nature, they also present specific performance factors such as those originated at application level due to creation pattern or task dependencies; as well as at library level due to task creation overhead, task scheduling policies, the amount of tasks created, and memory utilization per task. Some strategies can be applied, such as adapting dynamically the granularity of the task level parallelism, and adjusting the scheduler behavior to be aware of the application and hardware requirements.

3.3 Modeling performance and defining tuning strategies

A performance model identifies what is observed, and why is the problem originated. To define a performance model it is necessary to know the details of the performance factor, such as measurable causes (e.g. configuration values, thresholds, memory occupancy) and probable causes (e.g. data imbalance, false sharing). Generating a complex performance model is sometimes counterproductive because the dynamic analysis and data collection generates overhead.

It is also necessary to propose and evaluate the feasibility of applying previously defined strategies to tune the application, such as modifying the number of threads, the scheduling of loops, the distribution of tasks, and so on.

3.4 Evaluating the impact

The applied strategy has to be evaluated. Initially it can be implemented with manual instrumentation. The expected performance gain has to consider that the monitoring, the dynamic analysis, and the tuning strategies are generating overhead. Furthermore, it is necessary to provide certainty that the strategy can be dynamically instrumented.

The methodology has to iterate the performance factor evaluation, until the strategy coverage is considered appropriate. Understanding that, the performance factor could be relevant and the strategy could improve performance for other applications, another hardware system, or another manager library.

4. Tuning the number of threads: Case Study

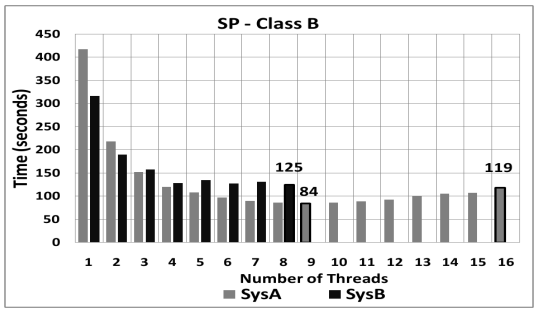
This section shows the application of the methodology on the Scalar Pentadiagonal solver (SP) application of the NAS Parallel Benchmark suite on two different architectures.

This case of study presents an existing performance factor in SP application. By following the methodology stages, the performance factor is detected in the analysis stage, a strategy is proposed to effectively tune the application, and finally, the impact of the strategy is evaluated.

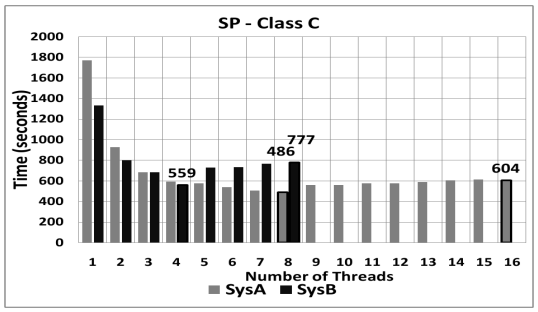
4.1 System characterization

The system characterization provides an insight on the specific context and has been determined following the template in Fig. 2.

- **Application**: SP benchmark of NPB-3.3.1 (OpenMP version), running class B and C configurations. Problem size for Class B is 102 elements and for Class C is 162 elements per matrix dimension. Both classes are configured with 400 iterations. For this application only data parallelism is used.
- **Manager library**: GNU libgomp 4.5.2, which implements the OpenMP 3.1 specification.
- **Hardware**; the following hardware platforms are available to evaluate the application.
  - SysA; SMP system with 2x AMD Opteron 6128 @ 2GHz processors with 8 cores; total amount of 16 cores and 32GB of main memory. Shared cache L1 of 64KB per core pair, Cache L2 of 512KB



(a) SP Class B execution times



(b) SP Class C execution times

Fig. 3: NAS SP Class C and Class D in SysA(16 cores) and SysB(8 cores)

shared by core pairs and shared 5MB of Cache L3 by groups of 4 cores.

– SysB; SMP system with 2x Intel Xeon E5430 @ 2,6GHz processors with 4 cores; total amount of 8 cores and 16GB of main memory. Dedicated Cache L1 of 32KB and Cache L2 shared by core pairs. Enabled hardware counters.

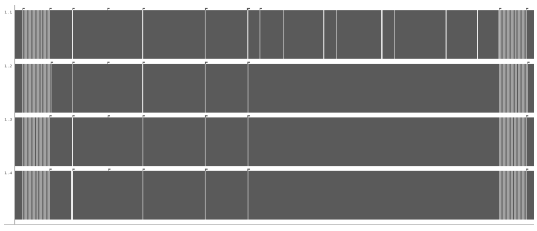
4.2 Analysis of performance factors

The NAS benchmarks have been characterized by executing them in different hardware platforms. There is a set of configurations to be evaluated, by using different data input types (NAS Classes) and by evaluating the benchmarks for different numbers of threads.

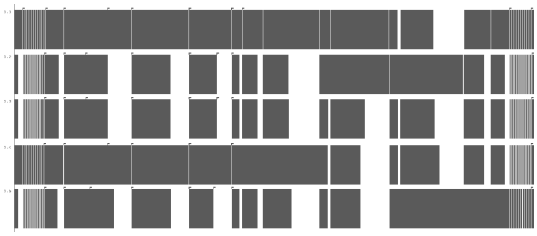
Result on Fig. 3(a) and Fig. 3(b) show a performance issue related with the ideal number of threads to be configured to achieve the best performance. This is because a performance factor is limiting the scalability.

Fig. 3 shows the results for different configurations in both systems. The evaluation of different data sizes and iterations represented by classes B and C into the hardware platforms SysA with 16 cores, and SysB with 8 cores. In all cases it can be seen that using half of the possible threads is achieving the best scalability.

To analyze the origin of the performance degradation, which appears when configuring more than half of the possible threads, it is necessary to compare critical configurations in detail. This analysis, described below is performed on SysB Class C.



(a) System B (4 threads) - 1 SP iteration



(b) System B (5 threads) - 1 SP iteration

Fig. 4: Paraver trace detail for one's SP application iteration on SysB

Table 1: SysB ClassC execution time (sec.) and cumulative percentage (relative to total time T_{ref}) of use for the weightiest parallel regions (x,y and z_solve, and rhs).

	Number of threads							
P.Reg	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
x_sol	175	87	60	**46**	85	88	103	**120**
y_sol	199	100	70	**53**	87	90	99	**106**
z_sol	224	112	79	**60**	94	96	105	**113**
rhs	537	327	331	**277**	328	342	330	**315**
T_{ref}	1331	797	683	559	725	732	765	777
%	85.2	82.8	79.3	78.3	81.8	81.9	83.2	84.4

Paraver [14] trace visualizer is used for a deeper analysis to compare the execution of 4 threads (559 seconds) and 5 threads (725 seconds) in the architecture SysB. Fig. 4 shows a thread imbalance because a repetitive pattern is observable for 10 iterations in the 4 thread configuration, an undetermined behavior appears for the 5 thread configuration. The dark blocks represent execution and the light blocks represent synchronization, scheduling, and idle states. Thread imbalance performance factor is shown for each iteration in the 5 threads configuration case. The thread imbalance can be measured in detail to find its cause.

The profile shows that the summarized imbalance for 4 threads total execution represents the 1%, and it is increased to 14% for 5 threads. ompP [15] provides the cumulative time spent in every OpenMP parallel region (P.Reg). By analyzing in detail the call graph, to refine the granularity analysis Table. 1 presents the overall percentage for the 4 main parallel regions. The performance degradation between the 4 thread execution in comparison with the 8 thread execution, of x_solve(160%), y_solve(100%), z_solve(88%) and the rhs(13%), is quite significant.

Thread imbalance is located mainly in four parallel re-

Table 2: x_solve parallel region on SysB ClassC for one iteration execution. Where T_{it_n} is the time for n-iteration and T_{ref} is the measured time for 400 iterations.

P.Reg	**Number of threads**							
x_solve	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
$\%C_{L2}$	1	1	1	1	**7**	**11**	**14**	**16**
T_{it_1}	0.45	0.25	0.16	0.13	0.22	0.22	0.27	0.3
T_{it_400}	180	100	64	52	88	88	108	120
T_{ref}	175	87	60	46	85	88	103	120

gions and, as it is shown in Fig. 4(a) when comparing the same class in different architectures, it is dependant on the system architecture. Therefore, an analysis of parallel regions have to consider hardware implications.

To analyze in detail the origin of the hardware performance factor, a representative function has been instrumented with PAPI [16] to obtain information from hardware counters (L2 cache misses percentage $\%C_{L2}$) for 1 iteration. The results on Table. 2 for the function x_solve show that the percentage of accesses to main memory increases significantly when the last level cache is shared between cores. This indicates that the threads are racing for memory in the highest level of cache, and consequently generating a significant overhead. Consequently, the amount of data generated for the classes B and C are exposing a performance factor related to memory occupancy, taking into consideration the core associativity from the hardware architecture.

The performance factor in the x_solve parallel region is originated by memory occupancy, when threads are sharing L2 cache modules the cache overhead degrades performance.

To conclude the analysis stage and to set an specific context for impact evaluation, the performance factors found in the analysis stage are, thread imbalance (summarized imbalance of 14% in 5 thread configuration), due to different memory access latencies, and a heavier performance factor due to memory bottleneck. As shown in the memory occupancy located at L2 cache in SysB for the x_solve function, which achieves 46 seconds in a 4 threads configuration compared with the 120 seconds for 8 threads configuration.

4.3 Modeling performance and defining tuning strategies

The performance factors exposed in the previous stage have been statically measured through the function execution time, and memory hardware counters. However, it is known that these metrics can be dynamically obtained by instrumenting the application using timers and the PAPI API.

The number of threads can be dynamically tuned by instrumenting the application and inserting in the appropriated point the *omp_set_num_threads(VALUE)* OpenMP function, that allows explicitly to define the number of threads to be created in the subsequent parallel regions.

Taking advantage of this previous knowledge, an initial coarse grained strategy is presented,

The SP application is iterative and those iterations are balanced because they execute in the same time. Then, it is possible to execute the N initial iterations with different configurations to acquire a prior knowledge of the performance. Next, the best configuration parameter is selected to tune the application for the remaining iterations. The SP NAS benchmarks are configured for 400 iterations in classes B a C. The number of iterations for N, used to acquire the best configuration, must be as small as possible.

The performance model proposed is based on selecting the number of threads configuration with the minimum time obtained in the characterization phase.

The strategy has been implemented and executed on SysB, the initial number of threads is configured to the maximum number of available cores, the measurement point is located at the beginning of the iteration. Finally, when the characterization phase is finished, the number of threads variable is tuned to the best selected configuration to achieve the minimum time for the remaining iterations.

4.4 Evaluating the impact

The evaluation of the strategy on SysB is as follows, the average execution time for the instrumented application is 562 seconds, which represents only an overhead of 0.5% compared to the 559 seconds of the original execution with 4 threads.

For this case of study the strategy has been implemented in the application source code. The overhead generated by instrumentation is less than 2 seconds.

Furthermore, the analysis of performance factors have shown that at basic blocks level there are other performance factors based on memory occupancy located at cache level 2 in SysB (evaluated for x_solve function), and also, a natural thread imbalance due to different core associativity of executing threads that achieve different memory access latencies in cores. Therefore, it is necessary to continue with the evaluation phase to define a strategy that achieves a higher impact on the application's performance.

5. Applying the dynamic tuning strategy

With the aim of demonstrating that dynamic tuning on OpenMP is feasible and to analyze the tuning overheads, we have implemented the strategy described in Section 4.

The strategy is implemented as a wrapper to libgomp library. The strategy has been applied by adding the control logic to the entry and exit points of the parallel regions, because parallelism on SP is defined by parallel regions and parallel loops with static scheduling. At low level compilation, those transformations have the same entry and exit point. Therefore, the tuning strategy has been implemented by hijacking libgomp function calls and replacing them with

the monitoring, analysis and tuning functions schematized below.

```
GOMP_parallel_start(params){
  if (characterization_Phase){
    config_Params = generate_Config();
    Start_Timer();
    real_GOMP_parallel_start( config_Params )
  }else{
    real_GOMP_parallel_start( tuned_Params )
  }endif  }

GOMP_parallel_end (){
  real_GOMP_parallel_end()
  if (characterization_Phase){
    Stop_Timer();
  }else{
    if(last_charact_phase( iter_Counter ))
       tuned_Params = time_Analysis();
  iter_Counter++;  }
```

The SP benchmark has been evaluated in different statically characterized architectures. The implemented solution is obtaining the best possible configuration by monitoring only the 5% of total number of iterations. The performance achieved in comparison with the original execution is presented in the table 3.

Table 3: Execution time (sec.) for the dynamic tuning strategy and execution without tuning for classes B and C. The tuning strategy uses 5% of total iterations for the characterization stage.

Class	Tune	SysB.	SysA.	Class	Tune	SysB.	SysA.
B	Dyn.	128	86	C	Dyn.	586	485
B	-	125	119	C	-	777	604
Speedup		0.97	1.38	*Speedup*		1.32	1.24

The strategy has been applied at iteration level. However, it is important to take into account the granularity of the tuning strategy to evaluate this strategy with real applications because it is possible to apply the strategy acting at parallel region level, achieving even greater performance improvements.

6. Conclusions

In this work, we have shown that, due to the growing heterogeneity of multicore systems, a way for improving the applications' performance is to dynamically adjust the applications' requirements to the characteristics of each hardware architecture using the code provided by the parallel programming framework.

A methodology has been proposed, with the aim of detecting performance factors, which can arise as performance problems depending on the context. The methodology must allow to detect performance factors on applications, moreover, for each detected performance factor it helps to define a performance model, the monitoring parameters, and tuning strategies that will allow to develop a dynamic strategy.

The methodology has been applied for a case of study based on the benchmark SP from the NAS PB suite. A performance factor has been presented, which is based on memory occupancy. Monitoring elements have been shown, which are reporting effectively the existence of the performance problem. Following the methodology, a strategy to dynamically tune the application has been implemented and evaluated, achieving a good performance in comparison with by default configuration.

The future work includes evaluating the strategy with a real application, and developing applied strategies for the rest of performance factors mentioned into the methodology.

Acknowledgment

This research has been supported by the MICINN-Spain under contracts TIN2007-64974 and TIN2011-28689

References

[1] D. Wentzlaff, et al., On-chip interconnection architecture of the tile processor, Micro, IEEE 27 (5) (2007) 15 –31.

[2] L. Dagum, R. Menon, Openmp: an industry standard api for shared-memory programming, Computational Science Engineering, IEEE 5 (1) (1998) 46 –55.

[3] R. Blumofe, et al., Cilk: an efficient multithreaded runtime system, SIGPLAN Not. 30 (1995) 207–216.

[4] C.-K. Luk, et al., Pin: building customized program analysis tools with dynamic instrumentation, SIGPLAN Not. 40 (2005) 190–200.

[5] G. Lee, et al., Dynamic binary instrumentation and data aggregation on large scale systems, International Journal of Parallel Programming 35 (2007) 207–232.

[6] A. Morajko, et al., Mate: Dynamic performance tuning environment, in: Euro-Par 2004 Parallel Processing, Vol. 3149 of LLNCS, Springer Berlin / Heidelberg, 2004, pp. 98–107.

[7] B. Wicaksono, et al., A dynamic optimization framework for openmp, in: OpenMP in the Petascale Era, Vol. 6665 of LLNCS, Springer Berlin / Heidelberg, 2011, pp. 54–68.

[8] S. Olivier, et al., Uts: An unbalanced tree search benchmark, in: Languages and Compilers for Parallel Computing, Vol. 4382 of LLNCS, Springer Berlin / Heidelberg, 2007, pp. 235–250.

[9] A. Duran, J. Corbalan, E. Ayguade, An adaptive cut-off for task parallelism, in: High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, 2008, pp. 1 –11.

[10] S. Olivier, J. Prins, Evaluating openmp 3.0 run time systems on unbalanced task graphs, in: Evolving OpenMP in an Age of Extreme Parallelism, Vol. 5568 of LLNCS, Springer Berlin / Heidelberg, 2009, pp. 63–78.

[11] D. Bailey, et al., The nas parallel benchmarks summary and preliminary results, in: Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on, 1991, pp. 158 –165.

[12] A. Duran, et al., Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp, in: Parallel Processing, 2009. ICPP '09. International Conference on, 2009, pp. 124 –131.

[13] J. M. Bull, D. O'Neill, A microbenchmark suite for openmp 2.0, SIGARCH Comput. Archit. News 29 (2001) 41–48.

[14] V. Pillet, et al., Paraver: A tool to visualize and analyze parallel code, Tech. rep., In WoTUG-18 (1995).

[15] K. Fürlinger, M. Gerndt, ompp: A profiling tool for openmp, in: OpenMP Shared Memory Parallel Programming, Vol. 4315 of LLNCS, Springer Berlin / Heidelberg, 2008, pp. 15–23.

[16] S. Browne, et al., A portable programming interface for performance evaluation on modern processors, International Journal of High Performance Computing Applications 14 (3) (Fall 2000) 189–204.

Efficient Runtime Algorithm Selection of Collective Communication with Topology-Based Performance Models

Takeshi Nanri[1,2] **and Motoyoshi Kurokawa**[3]
[1]Research Institute for Information Technology, Kyushu University, Fukuoka, Japan
[2]CREST, JST, Japan
[3]Advanced Center for Computing and Communication, RIKEN, Saitama, Japan

Abstract— *In communication libraries for recent supercomputers, decisions of appropriate implementations are becoming difficult by increasing size and complexity of them. Especially, because of the wide variety of the rank allocation and the improvability of the collisions, traditional static method cannot choose the appropriate technology for the given situation. As a dynamic technique for choosing implementation technologies, this paper introduces a method that selects a suitable algorithm of collective communications at runtime. At the first invocation of a collective communication, this method predicts the performance of each algorithm from the information about the rank allocation, the network topology and the routing policy. Then it discards the algorithms that are predicted much slower than others. After that, it examines each algorithm at each invocation of the collective communication to find the fastest one empirically. Results of some preliminary experiments showed the efficiency of the proposed method.*

Keywords: Collective Communication, Performance Model, Runtime Optimization

1. Introduction

In high-performance computing, communication libraries are playing an important role. To fulfill the strict requirement on performance with limited resources, they change the implementation technologies of some functions according to the situation. For example, most of the libraries prepare at least two protocols, Eager or Rendezvous, for transferring messages, and chooses one of them with a consideration of the tradeoff between the performance and the memory usage. Similar techniques are used for choosing algorithms for collective communications or deciding sizes of segments in pipelined communications. Previously, these choices of implementations have been done statically, in which they are selected according to the thresholds provided before execution. In most cases, benchmarks on some possible combinations of parameters, such as the number of ranks and the size of messages, are used to decide these thresholds [8], [9].

However, as the sizes and the complexities of computer systems for high-performance computing are increased significantly, such static strategy has become insufficient to enable efficient usage of the systems. One of the reasons is simply because the parameters to be considered has become large in number and wide in range. In addition to that, especially for communication libraries, usage of cost-efficient topologies of interconnect networks, such as Fat Tree, Mesh or Torus, has increased the difficulty on appropriate choice. On these topologies, there are some additional issues that affect the performance significantly, such as the collisions among independent messages, and the distance between the sender and the receiver. This means, even when the number of ranks and the size of the message are the same, best implementation technology can be different according to the situations only known at runtime, such as relative locations of ranks. Therefore, demands for the techniques to choose suitable implementations at runtime are increasing [1], [10].

In this paper, as one of them, a method is proposed for choosing the appropriate algorithm of collective communications at runtime. Collective communications, such as broadcasts, reductions and all-to-all exchanges, are popularly used in parallel programs of computational sciences to achieve productivity and performance portability. There have been many algorithms introduced for each of these collective communications. The method proposed in this paper is a combination of two approaches, performance prediction and performance measurement. At first, it gathers the information about the allocations of ranks, and applies them to the performance models of the available algorithms along with the information about the topology and the routing policy of the system to predict the time for completing the communication with them. By comparing the results, algorithms that are predicted to be significantly slower than others are discarded from the candidates. Then, each of the remained algorithms is examined one at each call of the collective communication to gather the empirical performance data. After all algorithms are examined, the fastest algorithm is chosen to be used for the rest of the calls. In this paper, this method is applied for Alltoall communication on a system with Fat Tree topology. The effect of the method is examined with some experiments.

2. Alltoall communication

As previously mentioned, collective communications are patterns of communications among a group of ranks to

copy, reduce or exchange data. This paper applies our method of runtime algorithm selection method to one of those communications called Alltoall, which is frequently used in Fast Fourier Transform or in matrix transposing. This communication involves every rank interchanging data among all the ranks at the end of the operation. Since each rank needs to send different data for each receiver, this is one of the most network-intensive collective communications used in parallel programs.

There have been many algorithms of Alltoall introduced to achieve better performance. For example, Simple Spread algorithm basically posts all receives and all sends, starts the communications, and waits for all communications to finish. Each rank proceeds with the communication sequentially and the order of the communications for node i is as follows: $i \rightarrow i+1$, $i \rightarrow i+2$, ..., $i \rightarrow (i+p-1) mod p$. This parallel algorithm is achieved using the non-blocking communication, such as MPI_Isend and MPI_Irecv in MPI(Message Passing Interfase). This means that the rank returns immediately after a function call. Thus, consecutive communications can overlap. However, it may cause severe collisions on a rank at which many messages have arrived at the same time.

To avoid such collisions, there are some algorithms that divide the entire communications in phases. For each phase, the targets of communications are carefully chosen so that each rank sends data to different rank. Ring algorithm, for example, uses the phase number as the distance of the target, while the Pairwise algorithm uses exclusive-or on the rank number and the phase number to decide the ranks to exchange data. These algorithms can be further classified by the method how they synchronize the phases. In one implementation, barrier synchronization is called before each phase, while there is another implementation that calls a barrier only once at the beginning of the algorithm. Another approach uses a light barrier in which senders wait for acknowledgement messages from receivers before sending data, like rendezvous protocol.

On the other hand, Bruck algorithm packs data for different receivers in one message to reduce the number of phases. At first, it rotates the data blocks on the rank i by a distance of i blocks both at the beginning and at the end of the algorithm rank. Then it sends messages in $\log(p)$ steps. In each step k, each rank sends to $(myrank + 2k)$ and receives from $(myrank - 2k)$, where $myrank$ represents the id of the rank. Since it requires redundant message transfer, this algorithm is mainly used for small message sizes.

These algorithms are only a part of the introduced ones for Alltoall communication. The relative speed of each algorithm depends not only on the traditional parameters such as the size of the message and the number of ranks, but also other factors such as the topology of the system and the location of ranks. The relative speed among algorithms changes according to not only traditional parameters such as the size of the message and the number of ranks, but also the topology of the system and the location of ranks. Therefore, static strategy cannot find the appropriate algorithm for given situation.

3. Effect of the Rank Allocation on the Performance of Collective Communication Algorithms

This section examines the effect of the rank allocation on the relative performance of the algorithms of collective communications by an experiment.

As the environment of the experiment, RICC (RIKEN Integrated Cluster of Clusters) at RIKEN is used. Each compute node of RICC consists of two processors of Intel Xeon X5570 2.93GHz and 12GB of memory. Each processor has four cores, and the processes are mapped to the cores one by one.

Compute nodes are connected by InfiniBand with Fat-Tree topology. It has two spine switches, and both are connected to all leaf switches. Figure 1 shows the topology, and the routing policy of the interconnect network of RICC. Each leaf switch is connected to each spine switch with two links, which means, there are four links in total from leaf to spine. The total number of compute nodes is 1024. 20 nodes are connected to each of the 51 leaf switches, and four nodes are connected to one. Each compute node has a unique number. As shown in the figure, contiguous numbers are attached to the nodes in a leaf switch; compute nodes on the 0th leaf switch have numbers from 0 to 19, those on the 1st leaf switch have 20 to 39, and so on. On the other hand, four links to spine switches are numbered from 0 to 3. At the transfer of a message to the node in other leaf switch, the node number of the target is divided by four, and the remainder is used as the number of the link to be used for sending the message upwards from leaf to spine, as well as downwards from spine to leaf.

On such topology, the balance of the usage ratio of the links depends on the rank allocation. Therefore, in this experiment, 128 ranks are allocated in five patterns as shown below to see the effect of the unbalanced usage on the performance of the collective communication algorithms.

Pattern 0:

> In each of 32 leaf switches, four ranks are allocated on the compute nodes with the node number of the multiple of four from the one with the smaller

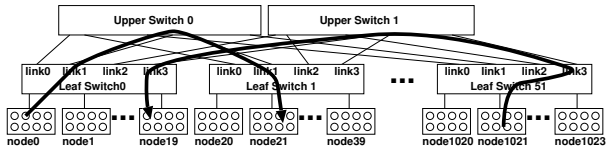


Figure 1: Topology and Routing Policy of the Network of RICC

number. That means in the 0th switch, four ranks are allocated on node 0, 4, 8 and 12, while in the 1st switch, another four ranks are allocated on node 20, 24, 28 and 32, and so on. With this pattern, at each leaf switch, four ranks share one link from, and to the spine switch.

Pattern 1:

In each of 32 leaf switches, four ranks are allocated on the compute nodes with the node number of the even number from the one with the smaller number. That means in the 0th switch, four ranks are allocated on node 0, 2, 4 and 8, while in the 1st switch, another four ranks are allocated on node 20, 22, 24 and 28, and so on. With this pattern, at each leaf switch, each two ranks share one link from, and to the spine switch. With this pattern, at each leaf switch, each two of four ranks share one link from, and to the spine switch.

Pattern 2:

In each of 32 leaf switches, four ranks are allocated on the compute nodes sequentially from the one with the smaller node number. That means in the 0th switch, four ranks are allocated on node 0, 1, 2 and 3, while in the 1st switch, another four ranks are allocated on node 20, 21, 22 and 23, and so on. With this pattern, each rank exclusively use one link from, and to the spine switch.

Pattern 3:

In each of 16 leaf switches, eight ranks are allocated on the compute nodes sequentially from the one with the smaller node number. With this pattern, at each leaf switch, each two of eight ranks share one link from, and to the spine switch.

Pattern 4:

In each of eight leaf switches, 16 ranks are allocated on the compute nodes sequentially from the one with the smaller node number. With this pattern, at each leaf switch, each four of 16 ranks share one link from, and to the spine switch.

Figure 2 and 2 show the elapsed time of each algorithm with the message size of 16KB and 1MB, respectively. Horizontal axis is the pattern number.

In the figure 2, the performance of the algorithm Bruck varies with the pattern. With pattern 0 and 4, the time with Bruck is more than two times longer than the time with the fastest algorithm. On the other hand, with pattern 2, the time with Bruck is almost the same with the fastest one. This is because of the difference of the message size transferred. When the number of ranks is P, Bruck repeats communication with message size 16KB * P/2 for $\log_2 P$ times, while other repeat 16KB message transfers for P-1 times. Therefore, change of the available bandwidth with the pattern has affected severer on Bruck than others. In addition to that, the reason for the difference of the performance

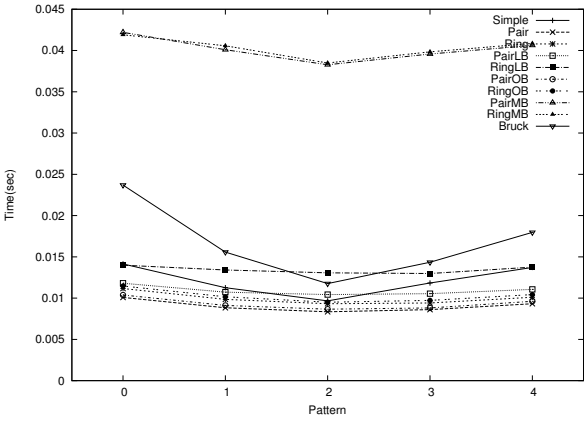


Figure 2: Elapsed time of each algorithm with each pattern(16KB)

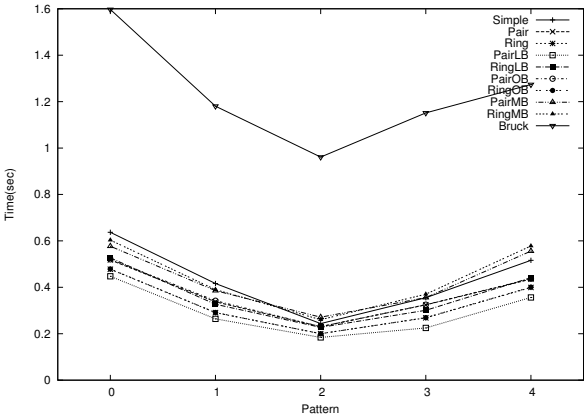


Figure 3: Elapsed time of each algorithm with each pattern(1MB)

between pattens 0 and 4, or patterns 1 and 3 is because the number of communications across leaf switches differs. In the figure 3, on the other hand, other algorithms also show the similar variation of performance with Bruck.

These results denote that the performance of algorithm changes significantly according to the relative locations of ranks. In addition to that, the effect of the rank allocation to the performance differs for each algorithm. Therefore, choice of the algorithm should be done with the consideration of the relative positions of ranks in the network topology.

4. Runtime Algorithm Selection of Collective Communication in Consideration of the Rank Allocation

4.1 Overview of the Method

The method proposed in this paper selects the algorithm according to the rank allocation as follows:

1) Before the execution of the program:
   Gather the information about the topology and the routing policy of the interconnect network of the system. Prepare performance models of each algorithm based on the information.
2) At the beggining of the execution:
   Acquire the information about the rank allocation.
3) At each call of a function of collective communication:
   If it is the first call of the function with the specified parameters, apply the rank allocation information to the performance models to estimate the time with each algorithm. Then, discard algorithms with the predicted time longer than the threshold from the candidate. This threshold is a relative value from the time of the algorithm that is predicted to be the fastest. The ratio of the time used in the experiments of this paper is 2.0.
   For each call, until all of the candidate algorithms are examined, use one algorithm for one call to complete the communication, and record the measured time.
   If sufficient number of measurements are done for each algorithm, choose the fastest one as the best algorithm, and use the best algorithm for the remaining calls.

In this method, the process for predicting performance of algorithms must be designed and implemented in consideration of the topology and the routing policy of the system. This paper introduces an example of the process of performance prediction for Alltoall communications on the Fat-Tree topology of RICC.

On the other hand, as the empirical technique for choosing the best one from the remaining algorithms, STAR-MPI [1] by Faraj, et al. is used. This method consists of two phases, Learning and Probing. In the Learning phase, one of the candidate algorithms is used for each call of the collective communication to measure the execution time with it. After repeated calls, when sufficient measurements are done on each algorithm, this phase is finished, and the fastest one is chosen to be used for the following the Probing phase. In this phase, the execution time of the chosen algorithm is measured for each call. If the time changed significantly from the previous measurement, it uses the second one.

4.2 Performance Prediction of Alltoall Algorithms on a Fat-Tree Topology

This section describes the performance prediction method for Alltoall algorithms on a Fat-Tree topology. In this method, the average bandwidth is estimated according to the topology and the allocations of ranks.

First of all, the estimation of the average bandwidth is done by considering the effect of collisions occur when each rank communicates with all the other ranks. A collision occurs when more than one communications share the same link with the same direction at the same time, The method proposed in this paper assumes that, when a collision occurs, the bandwidth of the link is shared evenly by each of the communications that caused it. Therefore, the average bandwidth of a link is estimated by dividing the base bandwidth of the link without collision, by the expected number of communications that use the link in the same direction.

In calculating this expected number on the link that connects leaf switches and spine switches, the way of calculation depends on the direction of communications. As for the upward direction, a link from a leaf switch to one of the spine switches is shared by communications that have the same value of the remainder after the division of the node number of the target by four. On the other hand, communications to the compute nodes in a leaf switch that share the same remainder of dividing the node number of the target by four share the same link from a spine switch in the downward direction. In this paper, the effects of the collisions in these two directions are assumed to be almost the same. Therefore, only the downward directions are considered to calculate the average bandwidth.

As the parameters for estimating the average bandwidth, N_{node} denotes the number of nodes, P_n is the number of ranks in a node, N_{LS} is the number of leaf switches, LS_i is the i-th leaf switch, UL_{ij} is the j-th link from LS_i to the spine switches, N_i is the number of nodes in LS_i, and NUL_{ij} is the number of nodes in LS_i with the remainder of dividing the node number by four is j. In addition to that, the base bandwidth of the communications in a node, in a leaf switch and across leaf switches is assumed to be the same value B.

The average bandwidth is estimated for each of the types of communications. For the communications between two ranks in a node, they assumed to be no collision, and the average bandwidth remains as B. On the other hand, for communications between two nodes in a leaf switch, one link from a node to the leaf switch is assumed to be shared by all ranks in the node. Therefore, the average bandwidth is calculated to be B/P_n.

For communications across leaf switches, as described above, the expected number of communications that share the same link is used to calculate the average bandwidth. For the link UL_{ij}, the expected number of ranks in LS_i that shares the link at the same time is calculated as follows:

$RCV_{ij} = 1 + (NUL_{ij} * P_n - 1) * (N_{node} - N_i) * P_n/(N_{node} * P_n - 1)$

The weighted average of these estimations of available bandwidth by using the ratio of the occurrence of each type of communication as the weight, is calculated in the following fomula.

$BRij = \frac{P_n - 1 + N_i - 1 + (N_{node} - N_i) * P_n / RCV_{ij}}{N_{node} * Pn - 1} * B$

BR_{ij} is calculated for each i, j with $0 \leq i < N_{LS}$, $0 \leq j < 4$, and the minimum value B_{min} is used as the average bandwidth of the system. This value is applied to the performance models of algorithms to predict the execution

time of collective communication with them. Table 1 shows the models of the algorithms of Alltoall used in this paper.

Table 1: Performance models of Alltoall

Algorithm	Model
Simple Spread	$(P-1)*L+(P-1)*M/B_{min}$
Ring	$(P-1)*(L+M/B_{min})$
Ring with One Barrier	$(P-1)*(L+M/B_{min})+(L*log_2P)$
Ring with MPI Barrier	$(P-1)*(L+M/B_{min})+(L*(P-1)*log_2P)$
Pair with Light Barier	$(P-1)*(L+M/B_{min})+L*(P-1)$
Pair	$(P-1)*(L+M/B_{min})$
Pair with One Barrier	$(P-1)*(L+M/B_{min})+(L*log_2P)$
Ring with MPI Barrier	$(P-1)*(L+M/B_{min})+(L*(P-1)*log_2P)$
Ring with Light Barier	$(P-1)*(L+M/B_{min})+L*(P-1)$
Bruck	$L*log_2P+P*M*log_2P/(2*B_{min})2$

These models use Hockney model as the basic model for point to point communication. In this model, time for each point to point communication is estimated as $L + M/B$ where M is the size of the message, and L and B are the latency and the bandwidth of the network respectively. This is one of the simplest algorithms. There have been some other models, such as LogP [2], LogGP [3] and PLogP (Parameterized Log-P) [4] that are proposed to achieve better correctness of the estimation. For example, PLogP represents the change of the bandwidth for different message sizes. Applying these models to the model of algorithms is remained as the future works.

In the implementation of Alltoall in this paper, L and B of the system are measured before executing the program. A benchmark program that repeats point-to-point communications is used for this measurement. In addition to that, the information about the topology, such as the number of links between leaf switch and spine switch, and the policy of routing are also collected before execution to construct the formula for calculating B_{min}.

On the other hand, the information of rank allocation is achieved at the beginning of the program. This information is applied to the formula to estimate the performance of each algorithm. From the results of this estimation, the algorithms that are predicted to be sufficiently slower than others are discarded from the candidates for the following Learning phase. The experiments in this paper used the threshold of time for discarding algorithms as the time two times longer than the fastest one. This threshold will be examined carefully with consideration of the accuracy of the prediction, in a future work.

5. Experiments

5.1 Overview

To examine the effect of the proposed method, experiments are done on a program called Alltoall Benchmark in the sample codes of STAR-MPI. This program is written in C with MPI, and repeats Alltoall communications for 200 times. The environment of the experiment is RICC. In the experiment, 10 jobs are submitted for each pair of a message size and a number of ranks.

On RICC, jobs are managed by a scheduler that is called Meta Job Scheduler. This scheduler prepares only one job pool, instead of job queues. The users specify the resource requirements at the submission of a job, such as the number of ranks, the size of memory and the estimated execution time. Then the scheduler uses these pieces of information to map the ranks of the job to the scheduling table of compute nodes. Nodes start execution of ranks according to the table. Since there is no job queue or node block, the scheduler can map ranks of jobs to the nodes without considering their locations. Therefore, with this policy, the number of idle nodes can be reduced, which can cause the throughput to be increased significantly.

This flexibleness on allocating ranks causes unstableness of the performance of communications. With this situation, existing static methods for choosing algorithms of collective communications cannot find the appropriate one. On the other hand, runtime selection techniques, such as the one proposed in this paper or STAR-MPI, are able to find suitable algorithms for given situations.

5.2 Results

Figure 4, 5 and 6 show the average times of alltoall communications with 16KB of message size on 1 rank $times$ 32 nodes, 4 ranks $times$ 32 nodes and 8 ranks $times$ 32 nodes, respectively. Horizontal axis is the job number. Each curve of the figures shows the average time with the method proposed in this paper (DYN GROUP), the method of runtime algorithm selection that measures all available algorithms (DYNþ£Őąþ£ŐąNOGROUP), and with the algorithms, Bruck (Bruck), Pairwise (Pair), Pairwise with Light-Barrier (PairLB), Pairwise with MPI-Barrier (PairMB), Pairwise with One-Barrier (PairOB), Ring (Ring), Ring with Light-Barrier (RingLB), Ring with MPI-Barrier (RingMB) and Ring with One-Barrier (RingOB).

As shown in the figures, performances of both DYN GROUP and DYN NOGROUP are close to the fastest algorith in most of the cases. These results indicates that both runtime selection methods could find the fastest algorithm. The overhead of each method is shown as the gap from the time of the fastest one. The major part of the overhead is the time for examining slow algorithms.

Figure 7 and 8 show the ratio of the time of DYN GROUP over DYN NOGROUP, for different number of ranks. In all

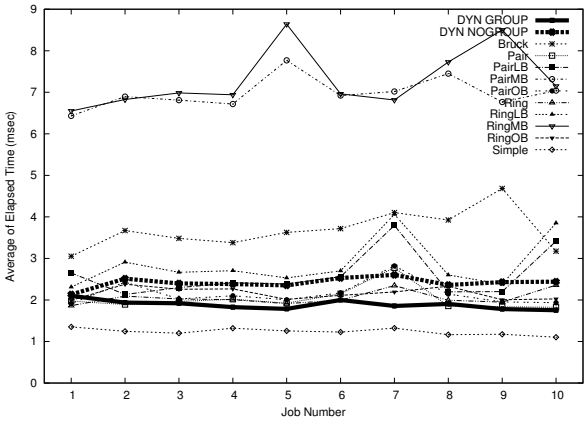


Figure 4: Average Time of Alltoall at each Job (16KB, 1 rank × 32 nodes)

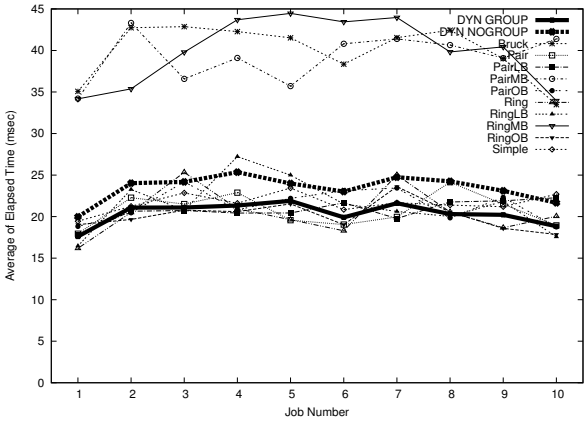


Figure 5: Average Time of Alltoall at each Job (16KB, 4 rank × 32 nodes)

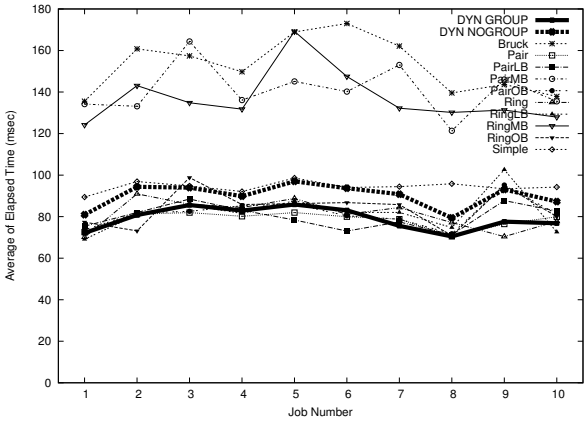


Figure 6: Average Time of Alltoall at each Job (16KB, 8 rank × 32 nodes)

the cases, DYN GROUP was faster than, or as fast as DYN NOGROUP. This is the effect of reducing the number of candidate algorithms at runtime. The reason of the significant effect on smaller message sizes is because most of the available algorithms are for middle or large sizes.

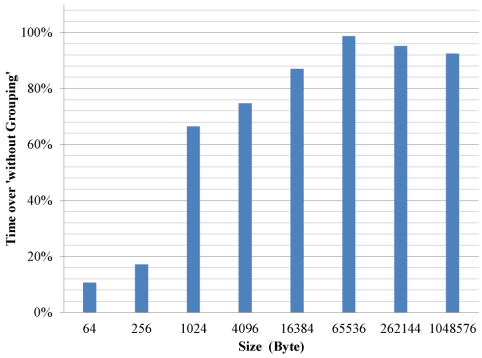


Figure 7: Ratio of the Time with and without Discarding Slow Algorithms (4 ranks × 32 nodes)

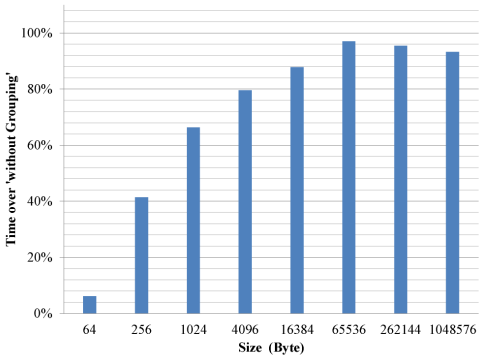


Figure 8: Ratio of the Time with and without Discarding Slow Algorithms (8 ranks × 32 nodes)

5.3 Accuracy of the performance prediction

This section examines the accuracy of the performance prediction introduced in this paper. As an example, Table 2 compares the predicted time with the average of the measured time of each algorithm in the case with 1 rank × 32 nodes.

The predicted time is less than a half of the measured time. Therefore, the prediction is not accurate enough. In the experiment, the proposed method has discarded two algorithms, Pair_mpi_barrier and Ring_mpi_barrier because they are predicted to be more than two times slower than the fastest one, Simple. However, the measured times shows that Bruck was also more than two times slower than Simple.

Therefore, the performance is expected to be gained if the prediction becomes more accurate. For example, us-

Table 2: Measured and Predicted Time of the Algorithms

Algorithm	Measured Time (ms)	Predicted Time (ms)	Remained or Discarded
Simple	1.352	0.606	remain
Pair	1.974	0.606	remain
Ring	1.865	0.606	remain
Pair_light_barrier	2.650	0.776	remain
Ring_light_barrier	2.310	0.776	remain
Pair_one_barrier	1.912	0.661	remain
Ring_one_barrier	1.989	0.661	remain
Pair_mpi_barrier	6.432	2.366	discard
Ring_mpi_barrier	6.551	2.366	discard
Bruck	3.053	1.150	remain

ing more detailed performance models, such as LogGP or Parametrized LogP, may solve this problem.

6. Related Works

Several algorithms for improving the performance of collective communications have been proposed for decades [5], [6]. More recently, some researchers have focused their efforts on taking advantage of existing algorithms in order to find techniques for selecting the most efficient algorithm for any given system/workload configuration. For example, Abdul Hamid, et al. have proposed an analysis of algorithm selection for optimizing collective communication with MPICH for Ethernet and Myrinet networks [8]. Instead of using the same thresholds for any MPICH installation on any parallel computer, they have investigated a way to find the optimum change-over points for different systems. Still their approach is a static optimization that could not be applicable on unstable situations by all means. Thus, Faraj, et al. have proposed STAR-MPI [1]. Their solution for tackling the best-performing algorithm selection problem uses dynamic profiling empirical data coupled with a static algorithm grouping method based on the performance results of different network platform systems. In addition, the authors also mentioned that the number of candidate algorithms can cause severe overhead, and performance models of those algorithms can be a clue to reduce their number. Nishtala, et al. have implemented a method of runtime algorithm selection on GASNet [10]. This method reduces the number of candidate algorithms by using performance models of them. However, since this method does not consider the effect of rank allocations, the estimation of the performance is not precise enough. In addition to that, this method measures the execution times of all candidate algorithms at the first call of the collective communication, overhead of the method is significant.

We have proposed an idea about combining both performance prediction and runtime measurement for choosing algorithms at runtime [11]. In this method, latencies and bandwidths are measured at the beginning of the program. Because the available number of measurement is limited to achieve low overhead, this method cannot consider the effect of rank allocation sufficiently.

7. Conclusions

A method for choosing an appropriate algorithm of a collective communication at runtime was proposed. This method used information about the network topology, the routing policy and the rank allocation to reduce the number of candidate algorithms examined at runtime. As an example of this method, Alltoall communication function was implemented for Fat-Tree topology of RICC. Experiments showed that the proposed method could find an appropriate algorithm with low overhead.

As the future works, functions for other collective communications and other topologies will be constructed.

Acknowledgements

Experiments in this paper are done on RICC(RIKEN Integrated Cluster of Clusters), at RIKEN, Japan.

References

[1] Faraj, A., Yuan, X. and Lowenthal, D.: STAR-MPI: Self Tuned Adaptive Routines for MPI Collective Communications, *Proceedings of International Conference on Supercomputing*, pp. 199–208 (2006).

[2] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R. and von Eiken, T.: LogP : Towards a Realistic Model of Parallel Computation, *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Pratice of Parallel Programming*, (1993).

[3] Alexandrov, A., Ionescu, M. F., Schauser, K. E. and Scheiman, C.: LogGP : Incorporating Long Messages into the LogP model - One Step Closer Towards a Realistic Model for Parallel Computation, *Proceedings of the 7th annual ACM symposium on Parallel Algorithms and Architectures*, pp. 95–105, (1995).

[4] Kielmann, T., Bal, H. and Verstoep, K.: Fast measurement of LogP parameters for message passing platforms, *International Parallel & Distributed Processing Symposium*, LNCS 1800, pp. 1176–1183, (2000).

[5] Barnet, M., Gupta, S., Payne, D., Shuler, L., van de Geijn, R. and Watts, J.: Interprocessor Collective Library, *Scalable High Performance Computing Conference*, pp. 357–364, (1994).

[6] Rabenseifner. R.: Optimization of Collective Reduction Operations, *International Converence on Computational Science*, LNCS 3036, pp. 1–9 (2004).

[7] Thakur, R., Rabenseifner, R. and Gropp, W.: Optimizing of Collective Communication Operations in MPICH, *Mathematics and Computer Science Division*, Argone National Laboratory, ANL/MCS-P1140-0304, (2004).

[8] Hamid, A. and Coddington, P.: Analysis of Algorithm Selection for Optimizing Collective Communication with MPICH for Ethernet and Myrinet Networks, *8th International Conference on Parallel and Distributed Computing*, Applications and Technologies, pp. 133–140 (2007).

[9] Sivac-Grbovi'c, J. P., Fagg, G. E., Angskun, T., Bosilca, G. and Dongarra, J.: MPI Collective Algorithm Selection and Quadtree Encoding, *In Proceedings of the 13th European PVM/MPI User's Group Meeting*, pp. 40–48 (2006).

[10] Nishtala, R.: Automatically Tuning Collective Communication for One-Sided Programming Models, PhDThesis, UC Berkeley, Berkeley (2009).

[11] Nanri, T. and Kurokawa, M.: Effect of Dynamic Algorithm Selection of Alltoall Communication on Environments with Unstable Network Speed, *International Conference on High Performance Computing and Simulation (HPCS), 2011*, pp. 693–698 (2011).

USING INTELLIGENT AGENTS FOR PERFORMANCE TUNING OF BIG DATA PARALLEL APPLICATIONS

Sherif Elfayoumy
School of Computing, University of North Florida, Jacksonville, FL, USA

Abstract - *Clusters of workstations are widely used to carry out big data applications. Applications running on these architectures may not produce the anticipated speedup because of imperfect load distribution. Developing intelligent application support tools that help improving performance by adjusting performance parameters is of great importance for the big data applications. A new architecture is presented in this paper to tune the performance of big data parallel applications with minimal effort from the application programmer and designer. This architecture is based on using intelligent agents to automatically and adaptively decide on load distribution with a focuses on image processing applications. This architecture is comprised of several features that are well-suited to heterogeneous environments, including: a flexible image partitioning scheme that assigns tasks in proportion to workstation's computational capabilities, an overlapping task assignment scheme that allows load balancing operations to be carried out with minimal communications overhead.*

Keywords: Big Data; Intelligent Agents; Performance Tuning; Parallel Computing.

1 Introduction

Many of today's applications are resource intensive with steadily increasing computational demands due to processing unprecedented amounts of data. As applications and data modality become more complex, new generations of hardware and software architectures are needed to serve the increasing demands of these applications. Achieving high-performance usually involves efforts on both hardware configurations and software architectures. The following sections introduce modern high performance computing architectures and the different classifications of intelligent agents as the core of next generation software.

Many hardware architectures have been developed through the last two decades to support high performance computing ranging from vector and superscalar processors to supercomputers and parallel computers. Supercomputers of commodity components are recognized today as one of the most efficient computer architectures that can provide an economical alternative for organization with big data processing needs [1]. Generally, clusters of workstations follow the distributed memory MIMD architecture, and usually achieve the required parallelism using direct message passing, or using software emulators to coherently share distributed memories [2]. This architecture has many advantages such as, no bus contention and no cache coherence. Commonly, processors exchange data by passing messages, which may result in some drawbacks such as increased latency and interruptions caused to the receiving processor to handle the received messages. Recognizing the concept of locality in designing parallel algorithms is recommended to minimize the number of messages traversing the interconnection network [2].

Performance instrumentation, prediction, analysis, and tuning play important roles in the design, development, and evaluation of big data applications. In parallel environments, the performance parameters space is much larger than that of the sequential case; consequently, dealing with performance in parallel environments tends to be more complex. The alternative of simulating various performance parameters is time consuming and for many large applications it may be impossible to accomplish in a reasonable amount of time. Numerous scattered efforts have concentrated on developing specific tools to help users instrument, analyze, predict, and tune the performance of their parallel and distributed applications using broad range of approaches. Most of these tools are run-time, using performance data gathered by instrumenting the application [5, 6, 7]. Unfortunately, for many reasons very few of these tools are actually being used [8]. A definition of the features needed to exist in performance analysis tools is provided by Reed et al. [9]. They emphasize that optimizing the behavior of complex parallel applications requires performance analysis tools to evolve, and replace postmortem analysis with real-time, adaptive optimization, and supporting high modality visualization. These new tools combined with big data application forms new paradigms of software architectures and we believe intelligent agents can be in the core of such tools.

Most of the performance problems in parallel applications are caused by imperfect load distributions. Developing intelligent agents to continuously monitor the load distribution and autonomously attempt to redistribute the load in a more balanced fashion should improve the overall application performance. Load-balancing techniques are distinguished by the timing of the load-balancing operations. Dynamic load balancing (run time) offers the most flexible and challenging load-balancing type because they provide for flexible scheduling policies that consider varying computing climates.

The problem of dynamically balancing load is typically divided into the following five phases: load evaluation, profitability determination, work transfer calculation, task selection, and task migration. Before load balancing is performed, it is necessary to establish that a load imbalance exists. A good load evaluation not only makes a good determination of the system load, but also does so with

minimal overhead measurement techniques. The load evaluation should also maintain some type of estimates associated with individual tasks to determine which tasks should be transferred to which workstation to balance the computation in the best possible way. Once it has been determined that a load imbalance exists, the cost of the imbalance should be contrasted with the cost of rebalancing. If the cost of the imbalance exceeds the cost of performing load balancing, then we should perform some load balancing. This comparison of the cost of imbalance vs. the cost of load balancing is the profitability determination.

The third phase involves the consideration of the measurements taken in the first two phases, to determine the work transfers necessary to balance the load. These considerations are used to generate a work transfer set. This set is the set of tasks that, once migrated, will rebalance the processing load. Next, in the task selection phase, we must select a set of tasks for exchange or transfer that will fulfill the work transfer set computed in the previous phase. When considering tasks for exchange or transfer, communication locality and task size should be considered. Thus, we should consider the cost of moving a task over a link, and the size of the transfer, since larger tasks will take longer time to migrate than smaller tasks. The final step in load balancing is to actually perform the exchange or transfer of tasks from one workstation to another. This step must be done carefully and correctly to ensure continued communication integrity and algorithm/program correctness. Additionally, if cost is not considered in this final phase, an excessive number of tasks can be transferred or exchanged, and the migration of these tasks will negatively influence system performance.

Due to the vast and varied number of parallel algorithms and the class of problems to which they are applied, there is no single dynamic load-balancing methodology that produces optimal results for all unbalanced workloads. Therefore, dynamic load-balancing strategies must not only be tailored to individual parallel run-time environments, but also tailored to the class of problems on which they operate. For these reasons, our solution is focused on big data applications, particularly those that process images.

A dynamic load-balancing approach for heterogeneous computing environments, DynamicPVM, was presented by Overinder [1]. DynamicPVM can offer speedups of 1.17 in light to medium load situations. While this approach can work in heterogeneous environments, tasks can only be migrated between similar architectures, which prevents it from becoming a true heterogeneous load-balancing approach. Furthermore, DynamicPVM utilizes frequent checkpoints in load-balancing operations causing computations to stall on all workstations during balancing operations.

Zaki and Parthasarathy presented several customized approaches to dynamic load-balancing [2]. The classes of applications that Zaki and Parthasarathy proposed customized approaches for include: matrix operations and TRFD from the Perfect Benchmark suite. Their work clearly demonstrates the effective use of differing strategies for different applications. However, all of the strategies they presented were developed for a homogeneous mix of workstations, rendering their results inappropriate for heterogeneous environments. Furthermore, each of their strategies relied upon synchronization points to perform load-balancing operations. These synchronization points can be a source of performance degradation on faster workstations that must wait on slower workstations to reach the synchronization point, making them even less suited to heterogeneous environments where performance capabilities offered by individual workstations can vary significantly.

Bozyigit presented a dynamic load-balancing scheme for heterogeneous networks of workstations that utilizes the previous execution histories of tasks to partition the current workload in such a manner as to minimize the need for rebalancing operations during computations [3]. While this method is suited to heterogeneous networks of workstations, rapidly varying load situations during computations remain unbalanced, resulting in sluggish load-balancing operations.

A comprehensive approach to dynamic load-balancing was presented by Watts [4]. This approach incorporates several features including: a diffusion algorithm that allows for balancing decisions to make trade-offs between work transfer and overall run time, and flexible task selection mechanisms that allows task movement to be governed by task size and communication costs. This approach has two serious drawbacks. First, the relative capabilities of each workstation are not considered when making load balancing decisions, making this approach poorly suited to heterogeneous environments. Second, this approach fails to balance the individual components that comprise the system load, which yields a low overall efficiency.

Most dynamic load-balancing strategies for rectifying load imbalances involve the migration of tasks from heavily-loaded workstations to lighter-loaded workstations. Hamdi and Lee presented a unique method for load balancing image-processing applications: instead of redistributing the tasks, they proposed the redistribution of the data [5]. This method would seem to incur large communication; however this approach is well suited to big data parallel applications such as image processing. Hamdi and Lee's approach is successful in rebalancing light and medium load imbalances, but offers poor performance when a large load imbalance exists between workstations. Furthermore, their approach partitions tasks into equal-size pieces for computation. This approach is not well suited to heterogeneous environments where the capabilities offered by individual workstations may vary significantly.

Hui presented a dynamic load-balancing system called DELAY [6]. DELAY incorporates a useful feature for dynamically load-balancing tasks executed on loosely-coupled architectures such as NOWs: the delay between load

measurement and reporting is considered when making load-balancing decisions. DELAY's scheduler prevents new tasks from being scheduled on a given workstation once the number of active jobs executing on it reaches a given threshold. DELAY requires the user to provide information regarding the relative capabilities of each workstation, making this approach poorly-suited not only to large networks of workstations, but also rapidly changing load scenarios.

Most of the investigated approaches are either limited to homogeneous environments, or are specifically tailored to a class of applications not congruous to image processing. Furthermore, these load-balancing approaches are designed not only to deal with unbalanced workloads resulting from non-uniform algorithms, but also load scenarios that change gradually and infrequently [5]. Practically, heterogeneous clusters of workstations can have load situations that vary frequently and dramatically.

2 Intelligent Agents

Software agents have evolved from multi-agent systems (MAS), which in turn form one of three broad areas, which fall under distributed artificial intelligence (DAI). The other two DAI areas are distributed problem solving (DPS) and parallel artificial intelligence (PAI). Hewitt [10] proposed the concept of the self-contained, interactive and concurrently executing object, which he termed *"actor"*. He defined an actor as "a computational agent that has a mail address and a behavior. Actors communicate by message passing and carry out their actions concurrently". Several definitions have been proposed to define an intelligent agent, although some of them are very specific to certain types of agents. Nwana [11] defines an agent as referring to a component of software/hardware that is capable of acting exactingly in order to accomplish tasks on behalf of its user. Despite the variety of definitions, there is general agreement that a good agent should possess most of the following features: intelligence, autonomy, taskability, awareness, a*ctiveability, flexibility, t*rustworthiness, adaptability, collaboration, and learning [12]. Nwana [11] has listed several dimensions to classify agents according to the features they exhibit, such as mobility, architecture, primary attributes, and role. Mobility is defined as the agents' ability to move around the network, so an agent can be classified as either mobile or static. According to architecture, agents are classified into three well known different classes of architectures, namely, the deliberative, reactive, and hybrid architectures. A minimal list of three primary attributes is developed. This list includes autonomy, learning, and cooperation. An agent may exhibit one or more of these primary attributes. Finally, the role that should be played by the agent is the last dimension for classifying agents. Examples of agents' classes in this dimension include, but not limited to, control agents and information agents that manage huge amounts of information such as the Internet search engines.

There are several more attributes to classify agents, but they are considered secondary to those mentioned. Out of the above classification, seven types of agents are identified: 1) collaborative agents, 2) interface agents, 3) mobile agents, 4) information/Internet agents, 5) reactive agents, 6) hybrid agents, and 7) smart agents [11]. Also, some applications combine agents of two or more of the above-mentioned categories. These agents are referred to as heterogeneous agent systems. The next section introduces an agent-based approach for parallel computing.

3 Agents-based Architecture

Although the primary motivation for using parallel computing platforms to carry out big data applications is their speedup potential, this potential is often difficult to realize in practice. Since many algorithms and development techniques can be used to build these applications, many system and performance parameters are involved. This causes difficulties in achieving the anticipated speedup without careful analysis and tuning of performance parameters. One objective of this research is to develop a new paradigm for adaptive performance tuning of big data parallel applications that is based on intelligent agents (performance agents).

The performance agent monitors the application behavior during run time, and adapts and tunes performance parameters' values (PPVs). The goal is for this performance agent to interact (collect, analyze, and decide) with the application as if it is an expert human user, so that performance is always improved, or at least not degraded. A general model for this approach is depicted in figure 1. The performance agent regularly receives the PPVs of each logical process (LP), and then it starts analyzing these values. Usually, the performance agent adapts and tunes these PPVs and finally sends the new values back to the corresponding logical processes. If the performance agent decided that no changes could improve the performance at this time, the application would continue using the same PPVs. Unfortunately, this may be difficult to achieve because many performance parameters are correlated, and this operation is expected to be time consuming. Thus, the agent uses heuristic techniques to find PPVs that lead to better performance.

This research recommends that a performance agent should exhibit intelligence, autonomy, adaptability, collaboration, and reusability. Also, it should act on behalf of the human user to make performance-enhancing adjustments to PPVs, thus showing intelligence. Although the agent can make independent decisions (autonomy), it can accommodate adjustments made by the user, thus showing adaptability. Moreover, a parallel application can have more than one agent. Each of these agents should perform a single job, such as a performance agent or a visualization agent. However, agents are not supposed to either overlap or compete. They can cooperate and exhibit collaboration. This performance agent needs to be generic such that it can be used with different parallel and distributed applications with different

topologies and underlying architectures, thus displaying reusability. Briefly, this paradigm is based on employing a performance agent that exhibits the above-mentioned features, to continuously improve the parallel applications performance.

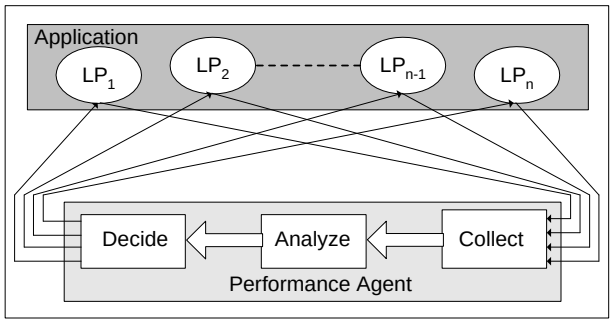


Figure 1. General Model

The architecture presented in this paper extends the approach presented by Hamdi by providing developing agents that are capable of balancing the load of processing a stream of images through the use of a flexible image partitioning scheme. Agents provide an adaptive load-balancing scheme by managing overlapping regions of computation. This architecture consists of one manager process agent, and a set of worker process agents. The manager process is responsible for queuing of the incoming image stream, image partitioning, transmitting data to workers, receiving completed computations from workers, and saving the processed images to disk. The manager process agent is responsible for balancing the load. The manager process flow diagram is depicted in Figure 2.

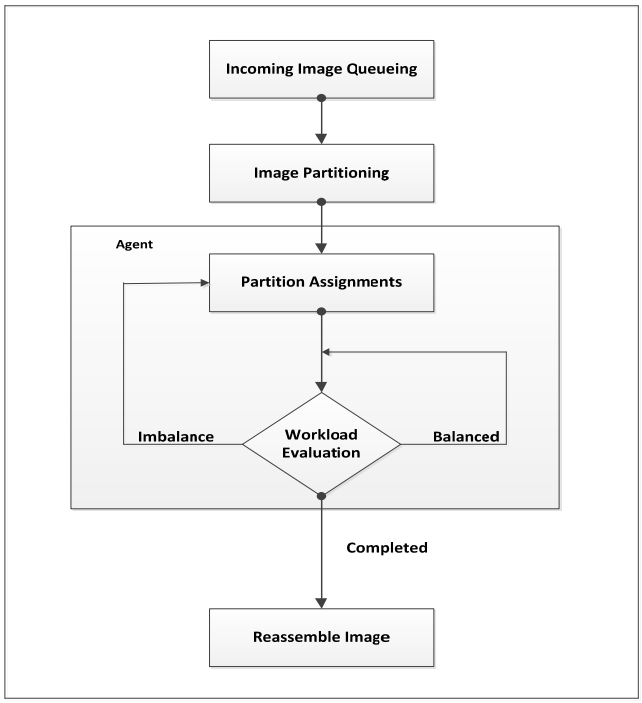


Figure 2: Manager Process Flow Diagram

Each worker process receives image data from the manager to process, processes image data, communicates local load information to the manager, and transmits processed image data back to the manager process. The worker process flow diagram is depicted in Figure 3.

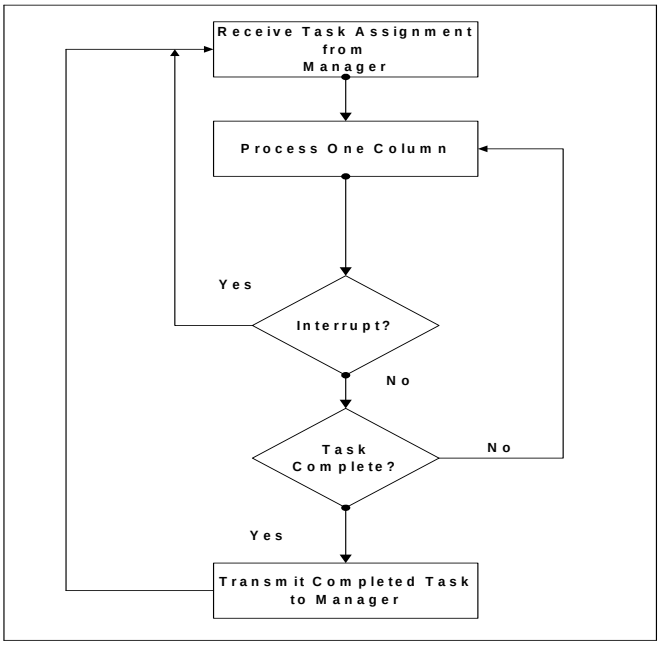


Figure 3: Worker Process Flow Diagram

This dynamic load-balancing architecture is designed for heterogeneous clusters of workstations, and first consideration was given to the load measurement scheme. In a heterogeneous environment, a load measurement scheme that eliminates the comparison challenges due to performance peculiarities of varying CPU and hardware architectures is desired.

3.1 Load Management

Hamdi proposed the use of the number of pixels processed at a worker to be a good load measurement [5]. This technique is adopted in our architecture for several reasons. First, count of pixels processed is a scalar value that presents no comparison challenges due to differing architectures. Second, a faster worker will be indicated by a large number of pixels processed when compared to other workers in the cluster. Additionally, the number of pixels processed is a load measurement that can be made by each worker with minimal impact on worker performance. Thus, we not only have a means of load measurement that presents no comparison challenges, but also whose measurement imposes little load on a worker.

3.2 Collection of Load Information

The manager process sends load interrupts to all worker processes. The frequency at which this occurs is to be determined experimentally. Upon receipt of a load interrupt, a worker transmits a vector containing the number of pixels it

has processed since the last load interrupt it received, and two additional scalar values that will be discussed later, to the manager. A worker resets its local pixels processed counter after it has transmitted its load measurement so that load measurements remain current, and do not reflect an execution history that is not commensurate with the current worker load situation.

3.3 Task Partitioning

Since the manager process in this architecture must collect load information from each worker process, the partitioning scheme should use this load information to assign tasks to workers in proportion to their historical contribution to the overall system throughput.

Assuming that we have n workers, let ℓ_i denote the load for worker *i*, where 0<=i<n-1. Recall that information is a scalar, the number of pixels processed. We compute the system throughput, or cumulative workload, in pixels processed per system time quantum (CW) using the following summation: load

$$CW = \sum_{i=0}^{n-1} \ell_i$$

The manager determines the percentage of columns in the image assigned to worker i,(A_i) by solving the proportion:

$$A_i = \frac{\ell_i}{CW}$$

Thus, each worker receives an image partition sized in proportion to its contribution of the current system throughput in pixels processed. In this way, image data is distributed in proportion to each worker's capabilities.

The manager process uses column wise block striping to partition incoming images. Each worker is assigned a partition based upon A_i. Next, we define a task vector that contains several scalar variables that the manager will transmit to each worker to describe the details of its next work assignment. The manager determines the leftmost and rightmost columns that worker i will be responsible for processing, designated LB_i and RB_i respectively. To offer reduced-cost load rebalancing operations, this architecture will transmit some columns in the image to two neighboring workers. By doing this, we eliminate data transmission requirements in remapping operations. With the existence of this "overlap" of image data, some load imbalances can be corrected with a reassignment of LB_i and RB_i on adjacent workers. The manager determines the amount of overlap that is transmitted to a given worker by first calculating the difference between the most and least-loaded workers in the cluster. We compute the degree of the largest imbalance (L_i) as:

$$LI = \max_{0<i<n}\{L_i\} - \min_{0<i<n}\{L_i\}$$

Next, the manager calculates the ratio of the imbalance (RI) to the current system throughput using: RI = LI/CW. The value of RI is used by the manager to help predict how much of an imbalance may need to be corrected during the next computation. The manager then calculates the number of columns to transmit as overlap (CO) to the workers using: CO = RI x width of image. Now that the manager has determined the number of columns that constitute the overlap transmitted to the workers, the manager must then determine how much of the overlap to distribute to each worker. The manager has already partitioned the non-overlapping data using A_i. Thus, the manager assumes that a lightly-loaded worker is more likely to assume responsibility for calculating more data in the overlap region to correct a future load imbalance. The manager uses A_i and CO to determine the number of overlap columns O_i to send to worker i using: $O_i = A_i$ x CO. The manager not only needs to calculate this data for later use during load-balancing operations, but also communicates this data to each worker. So, two new variables are added to the task vector previously introduced: LD_i and RD_i. LD_i and RD_i represent the leftmost and rightmost columns that worker i has sufficient data to process. The manager calculates the values of LD_i and RD_i for each worker using:

$$LD_i = LB_i - \frac{O_i}{2} \quad \text{if } 0<i<n-1$$

$$RD_i = RB_i + \frac{O_i}{2} \quad \text{if } 0<i<n-1$$

$$LD_i = LB_i \text{ and } RD_i = RB_i + O_i \text{ if } i = 0$$

$$LD_i = LB_i - O_i \text{ and } RD_i = RB_i \quad \text{if } i = n-1$$

After calculating each of these values for a given worker i, the manager process sends a task vector to each worker containing these calculated values. To maximize the possible benefit of utilizing the overlap in scenarios where a load imbalance exists, each worker will begin processing the columns farthest from the overlap region, with computations proceeding towards the local overlap region.

3.4 Load Evaluation and Balancing

As stated previously, the manager process periodically interrupts each worker involved in a computation to obtain load information. This load information is used to determine if a load imbalance exists, if it is profitable to correct the imbalance, and what steps should be taken to rebalance the load. After collecting l_i from each worker, the manager recomputes CW. The manager then computes the current ratio of throughput (Z_i) for each worker using: $Z_i = l_i/CW$. The manager then compares the values of Z_i to A_i for each worker i. By doing this, the manager can determine if the assignment ratio(A_i) is either too large or too small for a given worker's contribution to the computational workload based on Z_i. To make the determination that a load imbalance is large enough to warrant rebalancing, the manager sets a threshold (T), below which no rebalancing take place. Rebalancing decisions are based upon the following inequality: if ($|A_i - Z_i| > T$) perform rebalancing at worker i. Rebalancing is performed to satisfy the following inequality: $|A_i - Z_i| < T/2$. The right-hand

side of the inequality is divided by two so that the manager can make gradual changes to the distribution of work. The manager performs rebalancing by adjusting the values LB_i and RB_i for workers involved in the load imbalance. Specifically, faster workers are directed to take responsibility for one-half of their remaining overlaps adjacent to a slower worker, and slower workers are directed to forfeit responsibility for one-half of their remaining columns contained in the overlap with a faster adjacent worker. The manager accomplishes this by determining the new values of LB_i and RB_i using:

$$RB_{i-1} = RB_{i-1} + \left\lceil \left(\frac{RB_{i-1} - RD_{i-1}}{2} \right) \right\rceil$$

$$LB_{i+1} = LB_{i+1} - \left\lceil \left(\frac{LB_{i+1} - LD_{i+1}}{2} \right) \right\rceil$$

$$LB_i = RB_{i-1} \qquad RB_i = LB_{i+1}$$

Each worker maintains a local task vector that contains two additional entries: LP_i and RP_i. LP_i and RP_i are the leftmost and rightmost columns that worker i has already processed. These additional variables are used by the worker to manage the processing of the columns in the partition it receives from the manager, and to ensure that a worker does not attempt to compute columns outside its assigned partition. Additionally, these are the two additional values contained in the vector returned to the manager after the receipt of a load interrupt.

4 Conclusion

This agent-based paradigm has many strengths indicating support of system heterogeneity, minimal dependence on the application, reusability, and ease of use and development. The application programmer needs only to decide on the application performance parameters and metrics, insert instrumentation code in the application, and define the relations between the different performance parameters. The last task may not be an easy job in complex big data applications, so it is recommended that some heuristic techniques be used, which would be based on the application.

5 References

[1] A. Beguelin, J. Dongarra, A. Geist, P. Manchek, and V. Sunderam. "Solving Computational Grand Challenges Using a Network of Heterogeneous Supercomputers"; Proc. 5th SIAM Conf. on Parallel Processing, Philadelphia, 1991.

[2] V. Kumar, A. Grama, A. Gupta, and G. Karypis. "Introduction to Parallel Computing"; Benjamin and Cummings Inc., CA 1994.

[3] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam, PVM: Parallel Virtual Machine, The MIT Press, Cambridge, MA 1994.

[4] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra, MPI: The Complete Reference, MIT Press, Cambridge, MA 1998.

[5] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B. Schwartz, and L.F. Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment," *Proc. Scalable Parallel Libraries Conf.*, IEEE Computer Society, pp. 104-113, Oct 1993.

[6] K. Ekanadham, V.K. Naik and M.S. Squillante, "*PET: Parallel Performance Estimation Tool,"* Proc. 7th SIAM Conf. on Parallel Processing for Scientific Computing, pp. 826-831, 1995.

[7] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tool," IEEE Computer, vol. 28, no. 11, pp37-46, Nov 1995.

[8] A. Hondroudakis, and R. Procter, "An empirically derived framework for classifying parallel program performance tuning problems," Proc. SIGMETRICS symposium on Parallel and distributed tools, pp. 112-123, Welches, 1998.

[9] D.A. Reed, R.A. Aydt, L. DeRose, C.L. Mendes, R.L. Ribler, E. Shaffer, H. Simitci, J.S. Vetter, D.R. Wells, S. Whitmore, and Y. Zhang, "Performance Analysis of Parallel Systems: Approaches and Open Problems," Proc. of the *Joint Symposium on Parallel Processing (JSPP)*, pp. 239-256, Japan, Jun 1998.

[10] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," Artificial Intelligence, vol. 8, no. 3, pp. 323-364, 1977.

[11] H.S. Nwana, "Software Agents: An Overview," Knowledge Engineering Review, vol. 11, no. 3, pp. 1-40, Sep 1996.

[12] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng, "Distributed Intelligent Agents," IEEE Expert/Intelligent Systems and their Applications, vol. 11, no. 6, pp. 36-46, Dec 1996.

[13] R. Jain, The Art of Computer Systems performance analysis, Jain Wiley & Sons, 1991.

[14] Overeinder, B.J., et al.,"A Dynamic Load Balancing System for Parallel Cluster Computing," Parallel Computing and Simulation Group, University of Amsterdam, 1995.

[15] Zaki, Mohammed, "Customized Dynamic Load Balancing," High Performance Cluster Computing, vol. 1, 1997, pp 579-624.

[16] Bozyigit, M, "History-Driven load balancing for Recurring applications on networks of workstations," The Journal of Systems and Software, July 1998.

[17] Watts, Jerrell and Taylor, Stephen, "A Practical Approach to Dynamic Load Balancing," IEEE Transactions on Parallel and Distributed Systems, vol. 9, No. 3, March 1998.

[18] Hamdi, M, "Dynamic load-balancing of image processing Applications on clusters of workstations," Parallel Computing, vol 22, 1997, pp 1477-1492.

[19] Hui, Chi-Chung, "Improved Strategies for Dynamic Load Balancing," IEEE Concurrency, September 1999.

A method for scaling SPMD applications on Multicore Clusters *

Ronal Muresano, Dolores Rexachs and Emilio Luque
Computer Architecture and Operating System Department (CAOS)
Universitat Autònoma de Barcelona, Barcelona, SPAIN
rmuresano@caos.uab.es, dolores.rexachs@uab.es, emilio.luque@uab.es

Abstract— *Improving the performance of MPI applications on Multicore cluster is a huge challenge and even more, when we execute applications with high data synchronicity as is presented in SPMD paradigm. The different communications links of these clusters have to be managed properly if we wish to enhance performance metrics such as: efficiency, speedup and application scalability. In this sense, this work presents an approach that allows us to give an efficient solution for improving the strong and weak scalability under a defined efficiency for SPMD applications on Multicore clusters. This solution is focused on managing the communication heterogeneities using an analytical model, a mapping and a scheduling technique. The objectives are based on finding the maximum strong scalability point and determining how the problem size has to be increased in order to maintain an execution time constant while the number of core are expanded. The results obtained have demonstrated that using our method we can achieve improvements around 18% in the strong scalability. Also using our method, we have determined the ideal problem size, when we increase the number of core (weak scalability) and results show an small error of around (+/-) 4% in efficiency.*

Keywords: Strong and Weak Scalability, Performance, Efficiency

1. Introduction and Motivation

The efficiency and scalability are two key concepts when we are evaluating the performance in parallel applications. However, both performance metrics are seriously affected when we use heterogeneous execution environments such as Multicore clusters. These clusters are defined as heterogeneous because they integrate a hierarchical communication architecture, which can affect the performance. Therefore, if we consider the current use of multicore cluster and the communication heterogeneity in parallel computing, then, we have to design suitable strategies which allow parallel applications to be tuned with the aim of improving the application performance.

Also, the current integration of multicore nodes in high performance computing (HPC) has brought the inclusion of more parallelisms within nodes, enabling greater computing capacity within the node [1]. However, these capacity can be affected when we execute parallel applications due to parallel processes have to exchange information with others processes which can be located in the same node or in other node. These communications can be performed using the diverse communication paths, which are included in the hierarchical communication architecture. For this reason, communications have to be managed properly because they can present different speed and bandwidth, which may cause degradation and imbalance problems [2][3]. These problems can seriously affect when we use a pure MPI (message-passing interface) application with high data synchronicity.

Under this focus, one of the parallel paradigm, which is very affected by the communication heterogeneities is the Single Program Multiple Data (SPMD). This paradigm executes the same program in all MPI processes but uses different set of tiles to compute and communicate [4]. These tiles need to exchange information with neighboring tiles during a set of iterations. For this reason, when applications are designed under this paradigm, and these are mapped into multicore clusters, the programmer must consider the communication heterogeneity and how these can affect the performance.

Despite of these communication issues, we have to take into consideration that the multicore environments bring a huge computational capacity. For this reason, this work is focused on improving the performance of parallel application using a combination of different performance metrics such as efficiency, speedup and scalability (strong and weak).

Hence, the main objective of this work is based on how to design a method to manage the communication heterogeneity of multicore cluster in order to increase the SPMD application scalability under a defined efficiency. This method will allow us to determine the ideal number of cores that are needed in order to obtain a linear scalability using a constant problem size (strong scalability) while the efficiency is maintained over a defined threshold, and also, this method will permits us to determine how to maintain an isoefficiency system, where we can increase the number of processing elements (cores) and problem size while the execution time remains approximately constant (weak scalability).

On multicore clusters the tiles are computed in a similar time due to the homogeneity inside the core but the communication time of tiles can be performed using different communication paths and their communication times can be

* This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974 and the support of Julish Supercomputing Center.
*Contact Autor: R. Muresano, rmuresano@caos.uab.es
†This paper is addressed to the PDPTA conference.

totally different. In some cases, the differences between each link for a defined packet size can include up one and a half order of magnitude in latency. These variances are translated into inefficiency, which decreases performance metrics of SPMD applications.

As a first approach to manage the inefficiency, we have created a method, which manages the communication latencies using characteristics of application (e.g communication and computation ratio). This method is organize in four phases characterization, tile distribution model, mapping and scheduling and the main objective of this method is to find the maximum speedup while the efficiency is maintained over a defined threshold [5]. Taking advantage of this method, we have adapted it with the aim of controlling the hierarchical communication architecture with the objective of finding the maximum scalability point for both strong and weak scalability under a defined efficiency.

In this sense, we have introcuded the concept of supertile (ST). An ST is a unit which integrates a set of tiles where these tiles are divided in two types internal and edge. The main idea of these STs is to create a structure, which is assigned one per core. These STs manage the communication heterogeneity and also eliminate communication wasting time of parallel execution. This method takes advantage of the communication time assigning more computation tiles and hiding the communication effects. The division of STs among internal and edge allow us to apply an overlapping technique, where is overlapped the internal computation time while the edge communication is performed. The ST size is calculated considering the slowest communication path, allowing us to manage the communication between all links in the hierarchical communication architecture.

Then, this method is divided in two direction: the first one is to obtain the problem size and determine the ideal number of core which allow us to find the maximum strong scalability point under a defined efficiency threshold and the second direction is based on increasing the problem size according to the ST size allowing us to obtain a linear weak scalability. Both uses of the ST and the advantages obtained will be demonstrated in the experimental evaluation.

Finally, this paper is structured as follow: the related works are described in section 2. Section 3 presents of the hierarchical communication architecture and the performance effects over SPMD applications. Section 4 exposes the method for improving the application scalability and section 5 illustrates the performance evaluation. Section 6 draw the main conclusions and future works.

2. Related Works

Improving the performance of applications on multicore clusters has been widely studied. One method was developed by Liedbrock [6] which defines a manner to calculate a performance model of hybrid applications. This method was based on studying the scalability and fidelity of application on a specific multicore cluster. Another method for placed efficiently MPI process was designed by Mercier et al [7] where they studied how to establish an adequate placement policy for improving the performance. Another method was developed by Vikram [8]. This method seeks to improve the performance on multicore using mapping and scheduling strategies, using the management of communication links. On the other hand, our method attempts to enhance the performance of SPMD application on multicore through the combination of scalability and efficiency. The goal is to obtain the maximum speedup under a defined efficiency.

However, developing strategies for improving performance on heterogeneous communications environments is a challenge as was demonstrated by Oliver [9]. In this sense, there are works that are focused on proposing solution for managing the internal communication of the multicore node [10][11]. In this sense, our methodology takes advantage of the mapping scheduling and overlapping techniques in order to to create a novel method which improves the efficiency, speedup and scalability of the SPMD applications.

3. Heterogeneus Communication links and their performance effects

One important problem of executing SPMD application on Multicore is related to tiles have to exchange their information each iteration with the aim of calculating values for next iteration and this communication can create imbalaces issues that affect the performance. Also, the different communication patterns can vary according to the objective of the SPMD applications. In some cases, the communications can be executed in two, four, six, etc. bidirectional communications. These communications pattern are established at the beginning of the SPMD application and are kept until the application finishes.

Also, SPMD applications used to apply our method have to be designed under the characteristics of static, local and regular, where static, defines the communication pattern and this cannot vary during the execution, local communication, that determines the neighboring communication and it is maintained in all the execution, regular, because communications are repeated for several iterations and finally n-dimensional grid applications (however this characteristic has been evaluated using applications until 3 dimension). There are different kind of benchmarks and applications of diverse fields that accomplish all these characteristics. One example of benchmark can be detailed the NAS parallel benchmark in the CG, BT, MG, SP [12], all these benchmark have been designed for 2D and in some case for 3D grid problem size. Also, there are examples of real applications especially simulation problem such as fluid dynamics, heat transfer, etc.

A real example of this phenomena can be detailed in figure 1, where is execute a (LL-2D-STD-MPI) application of the

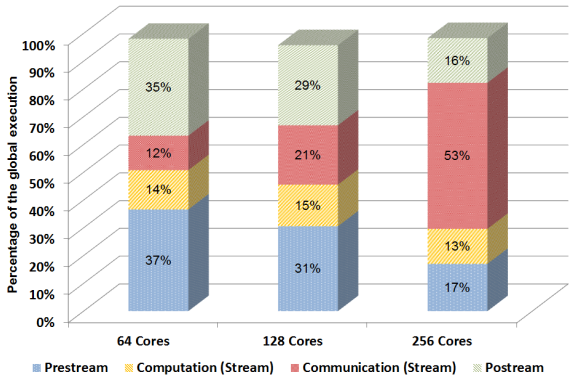


Fig. 1: Functions decomposition of LL-2D-STD-MPI.

mplabs suite over a juropa cluster [1] and results show the impact of communications over the total execution. In this case, for 64 core the impact of communication is around of 12,42% of total execution. However, when we increase the number of core to 256 for the same problem size the communication is around 52,59%. This example illustrates a clear impact of the communications on the application, where more than a half of application is communicating and hence the scalability and efficiency get worse.

Under this scenario, we can propose a method that allows us to manage the communication using the Supertile (ST) definition, where a ST is a structure composed by internal and edge tiles with the aim of allowing us to compute the edge tiles and then to overlap the edge communication with the internal computation. The problem of finding the optimal ST size is formulated as an analytical problem, in which the ratio between computation and communication of the tile has to be founded with the objective of determining the ideal size that maintain a close relationship between internal computation and edge communication. The ST size is calculated with two objectives: the first is to find the maximum strong scalability point under a define efficiency and second is to increase the problem size according to the ST size in order to maintain the execution time constant, when we increase the number of cores(weak scalability).

4. Method for improving the application scalability

This methodology is focused on managing the different communication latencies which are presented on multicore clusters due to the hierarchical communication architecture. The main idea is to develop a method that allows us to hide the communication effects that affect seriously the SPMD application. This method is integrated by four phases: characterization, tile distribution model, mapping and scheduling and all these phases allow us to obtain an efficient and a faster parallel execution [5].

[1]Cluster located in Julich Supercomputing center and it has 2248 computing nodes with 8 cores each node (17984 cores

As a first approach this method was designed to find the maximum speedup while the efficiency is maintained over a defined threshold, however, this method calculates the supertile size, which an import key for the scalability analysis. In this sense, our method has been updated with the aim of using the ideal number of cores and ST size that permit us to accomplish the objectives stated before.

4.1 Characterization phase

This phase is focused on performing an application and environment analysis with the aim of obtaining the application parameters which are used to calculate the analytical model. The main idea is to find a nearest relationship between the machine and the SPMD application. The parameters are classified in three groups: the application parameters, parallel environment and the defined efficiency.

The parameters determined allow us to establish the communication and computational ratio time of a tile inside of the hierarchical communication architecture. This relationship will be defined as $\lambda(p)(w)$, where p determine the link where the communication of one tile to another neighboring tile has been performed and w describes the direction of the communication processes (e.g. Up, right, left or down in a four communications pattern). This ratio is calculated with equation Eq. 1, where $Commt(p)(w)$ determines the time of communicating a tile for a specific p link and the Cpt is the value of computing one tile on a core. This characterization process has to be done in a controlled and monitored manner.

$$\lambda_(p)(w) = CommT(p)(w)/Cpt \quad (1)$$

4.2 Tile distribution Model

The analytical model for predicting the strong and weak scalability begins by considering the problem size and the number of cores that execute the application. For this reason, equation 2 represents an easy manner to calculate the ideal number of cores, where is divided the problem size (defined by M^n) by the ideal supertile size (K^n) [2], where n is the application dimension(e.g 1,2,3, etc). In this case, we are working on the strong scalability, where we define a problem size fix and we increase the number of core [13][14].

$$Ncores = M^n/K^n \quad (2)$$

Beside of the strong scalability, the equation 2 allows us to determine the behaviour of the weak scalability, where the number of cores and the problem size are expanded [15][16]. This equation is used to calculate the problem size M^n through the multiplication of the $Ncores$ by the supertile size K^n. However, to calculate both scalabilty we have to know the value of K, which depends on the overlapping strategy. In this sense, the equation 3 represents

[2]The supertile consist in a structure designed to allows us to overlap and hide the communication effects

the execution of an SPMD application using the overlapping strategy, where is first calculated the edge tile computation time $EdgeComp_(i)$ and then we add the maximum value between internal tile computation $IntComp_(i)$ and edge tile communication $EdgeComm_(i)$. This process will be repeated for a set of iteration $iter$. A first approach of this model can be found in [17]. However, the previos model was defined only for a specific number of dimension, and in this case we have defined for a n dimesnsional SPMD application and for studying strong and weak scalability.

$$Texe_{(i)} = \sum_{i=1}^{f} EdgComp_{(i)} + Max\begin{pmatrix} IntComp_{(i)} \\ EdgeComm_{(i)} \end{pmatrix} \quad (3)$$

$$EdgComp_{(i)} = (K^n - (K-2)^n \quad (4)$$

$$IntComp_{(i)} = (K-2)^n \quad (5)$$

$$EdgeComm_{(i)} = K^(n-1) * Max(CommT_{(p)(w)} \quad (6)$$

From the foregoing, the next step is to find the value of K considering the overlapping strategy between intenal computation and edge communication. The equation 7 shows how both values(internal computation and edge communication) can be equalized with the aim of finding the value of K. However, the edge communications are in function of the $CommT$. For this reason, we need to equalize the equation in function of Cpt. This is achieved using the equation 1. Having both internal computation and edge communication in function of Cpt, the next step is to find the value of K, replacing all the values in equation 7. Depending on the dimension of the SPMD application, we can obtain for example an cuadratic equation, cubic equation, etc.

$$K^{(n-1)} * max(\lambda_{(p)(w)} * Cpt) = ((K-2)^n/effic) * Cpt \quad (7)$$

4.3 Mapping phase

The main purpose of this phase is to apply a distribution of ST in the execution core. The ST assignations are made applying a core affinity which allows us to allocate the set of tiles according to the policy of minimizing the communications latencies (cite). This core affinity permits us to identify where the processes have to be allocated and how the ST have to be assigned to each core. However, the ST assignations should maintain the initial considered allocation used in the characterization phase.

This phase is divided in three key points. The first point performs a logical processes distribution of the MPI processes. The second function is to apply the core affinity, and the last one is the division and distribution of the STs. The mapping has to divide the tiles in order to create the ST considering the value of K obtained by the analytical model. It's important to understand that an incorrect distribution of the tiles can generate different application behaviors.

4.4 Scheduling phase

The main function is to assign a execution priority assignment to each tile with the aim of applying the overlapping strategy. This process establishes the highest priorities for tiles which have communications through slower paths and slower priority to internal tiles. This phase peforms an overlapping strategy, which is the main key of our method.

5. Performance Evaluation

In order to evaluate the maximum strong scalability under a defined efficiency and how the problem size and the number of cores (weak scalability) have to be increased with the aim of maintaining an execution time constant, we have used different multicore clusters composed by 16-17664 cores. However, this article has been validated using a large scale system which is setup with 2208 nodes with 2 intel Xeon X5570 nehalem-EP quad core processor (17664 cores), 24 GB of ram memory by node, 8 MB of l2 cache and infiniband interconnection network. This clustes is called JUROPA and it is located in the julich supercomputing center. Also, this methodology has been validated with different set of benchmarks and applications, but in this case we have used two examples of application of fluid dynamics integrated in the MPlab suite (LL-2D-STD-MPI and ZSC-2D-STD-MPI). Both versions are two dimensional problems and they accomplish the characteristics defined of local, regular and static.

The first application LL-2D-STD-MPI is integrated by 3 main modules: prestream, stream and poststream. The prestream and poststream are computation functions, where there are not any communication, while the stream function integrates the communication exchanging process between neighboring tiles. An example of this decomposition is illustrated in figure 2, where we can observe the impact of each function and the effect over the execution time. This example shows the relationship for a defined problem size of 2000x2000 using 1000 iterations (size that will be used for both applications). As can be detailed, the functions prestream and poststream maintain the same proportion of the execution time, while the stream begin to increase, when the number of core increases and the communication starts to introduce a strong impact over the execution (this was illustrated in figured 1). The communication in this case is around 53% of the total execution time for a 256 cores.

Another example is illustrated in figure 3, where is analyzed the ZSC-2D-STD-MPI application which integrates two exchanging functions called collision and stream. Both functions increase the inefficiency, while the number of cores is increased due to the communications. In this case, the impact of communication is increased while the number of cores is expanded (Fig. 3), where half of the time, the application is communicating when execute with 256 cores.

This scenario allows us to apply our method with the aim of improving the performance for both strong and weak

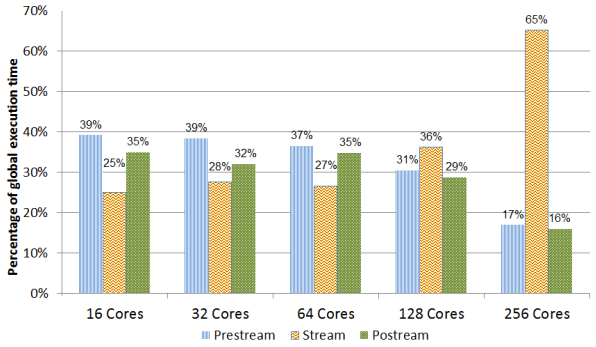


Fig. 2: LL-2D-STD-MPI application behavior.

Table 1: Tile distribution model and Supertile Calculation

Application	Problem Size	Effic	ST size	N Cores
LL-2D-STD-MPI	7200	90%	318	513
ZSC-2D-STD-MPI	7015	95%	155	2048

scalability. The first step is to characterize the application and the environment with the objective of finding the communication and computation ratio. In this sense, the tables included in figure 4 shows the ratio obtained using all the communication links in the juropa cluster and for both applications. Also, figures 2 and 3 illustrates the latency values, which can vary according to the communication level and also, they allow us to determine the slowest communication path.

The next step is to determine the ideal value of the ST, which will be used to both strong and weak scalability analysis. In this sense, we have defined the problem size and the desired efficiency, which we wish to apply our method. Using both problem size and efficiency together to the characterization values, we can calculate the ideal supertile which maintains the maximum strong scalability under a defined efficiency. These values can be detailed in table 1 and are calculated using the equations 2 and 7.

The first validation is related to the objective of determining the ideal number of cores that are needed in order to obtain a linear scalability using a constant problem size (strong scalability), while the efficiency is maintained over a defined threshold. This number of cores has been shown

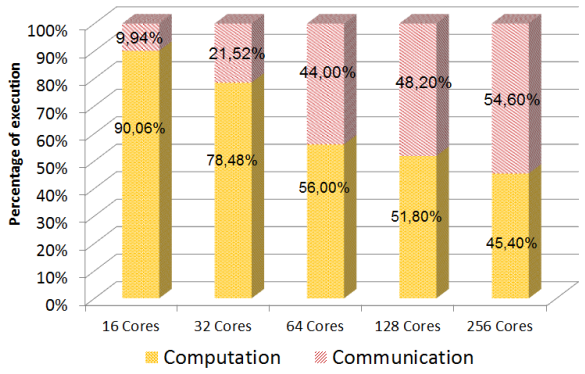
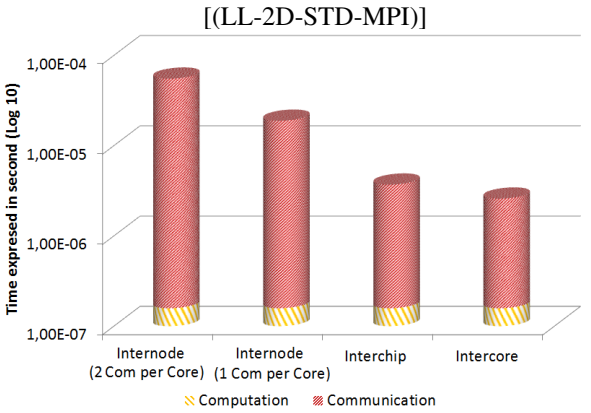
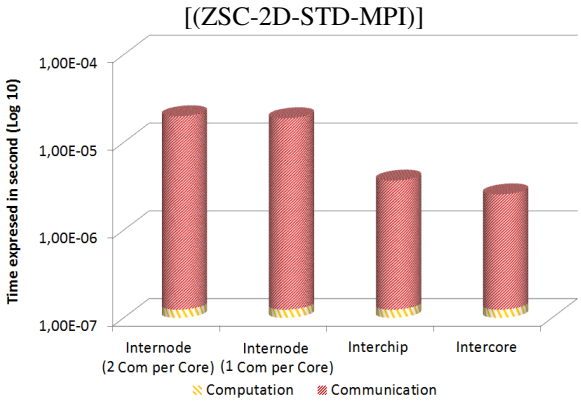


Fig. 3: ZSC-2D-STD-MPI application behavior.



Communication Link	Communication Time	Computation Time	Ratio λ(p)(w)
Intercore	2.34 E -06	1.58 E-07	15.38
Interchip	3.54 E-06	1.58 E-07	22.41
Internode (1 Comm)	1.87 E-05	1.58 E-07	118.35
Internode (2 Comm)	5.51 E-05	1.58 E-07	348.73



Communication Link	Communication Time	Computation Time	Ratio λ(p)(w)
Intercore	2.42 E -06	1.24 E-07	19.52
Interchip	3.19 E-06	1.24 E-07	25.73
Internode (1 Comm)	1.67 E-05	1.24 E-07	134.68
Internode (2 Comm)	1.97 E-05	1.24 E-07	158.87

Fig. 4: Communication and computation characterization

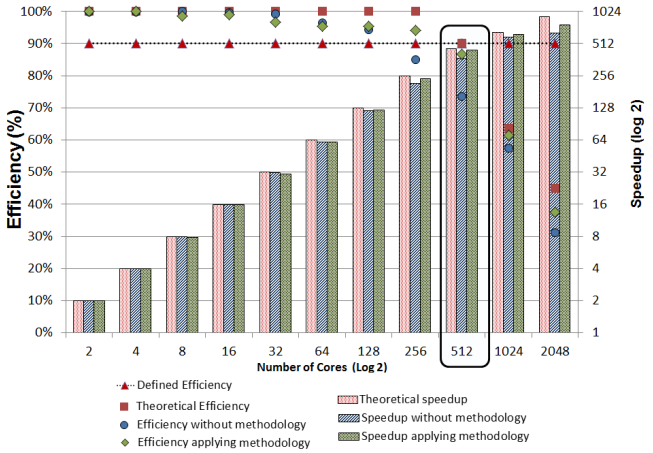
in table 1. Using these values, we have analyzed the scalability and efficiency of both applications. These have been performed evaluating the speedup and efficiency parameters using and not using our method with the aim of evaluating the improvements obtained using our method. Also, we calculated the theoretical behavior of applications according to the overlapping strategy between internal computation and the edge communication defined in our method.

In this sense, the figure 5 shows the behavior of speedup and efficiency of LL-2D-STD-MPI application, where the theoretical values calculated using our model have a similar growth until the ideal number of cores calculated for a defined problem size. Likewise, the application using the methodology has a close behavior than the theoretical with a small error rate around +/- 5% until the ideal number of cores and ST calculated in table 1. From that point, the error begins to increase motivated to the communication congestions. In this case, the edge communication times are greater than the internal computation and therefore, the

efficiency is seriously affected. Also, as can be illustrated in figure 5, the application without using our methodology begins to decrease considerably from 256 cores because the communication are not managed and they generated delays between each iteration that affect the scalability.

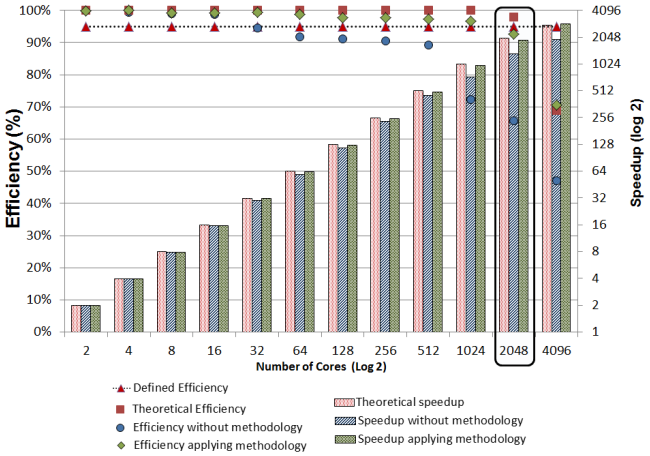
When we analyses both performance metrics with the objective of determining the maximum strong scalability under a defined efficiency, we can see in the table inside the figure 5, that speedup grows in both version. However, the efficiency has been defined in a threshold of 90% and this condition is nearly accomplished in the 512 cores for the application using our method. The efficiency and speedup improvements between the version using our method and not using our method are around 18%. This value is found when we divide the (speedup or efficiency) obtained using our method between the (speedup or efficiency) achieved without applying our method.

In similar manner, the figure 6 illustrates the behavior of ZSC-2D-STD-MPI application. the version using our method is above the threshold of efficiency until the number of cores ideal calculated with the model (Table 1). Otherwise is presented in the version that not use our method where efficiency is bellow the threshold from 64 cores. In this example, the error rate between the theoretical value and the value obtained using the version with our method is around 3,5 %. Also, the efficiency and speedup of the version without applying our method starts to decrease considerably from 1024 cores, while the version using our method begins to decrease from 4096 cores. In addition, can be detailed in figure 6 that speedup increases until 4096 core for both cases but the efficiency constraints defines the ideal number of cores that has to be used in order to find the maximum



Ncores	Efficiency without applying the methodology	Efficiency applying the methodology	Speedup without applying the methodology	Speedup applying the methodology
64	96,41%	95,40%	61,70	61,055
128	94,31%	95,31%	120,71	121,997
256	85,04%	94,05%	217,69	240,772
512	73,47%	86,77%	376,16	444,249
1024	57,53%	61,40%	589,13	628,779
2048	31,26%	37,45%	640,22	767,018

Fig. 5: LL-2D-STD-MPI strong scalability analysis.



Ncores	Efficiency without applying the methodology	Efficiency applying the methodology	Speedup without applying the methodology	Speedup applying the methodology
128	91,08%	97,63%	116,58	124,969
256	90,45%	97,67%	231,56	250,035
512	89,20%	97,19%	456,72	497,590
1024	82,58%	96,60%	845,58	989,226
2048	71,62%	92,64%	1466,74	1897,367
4096	53,21%	70,64%	2179,68	2893,485

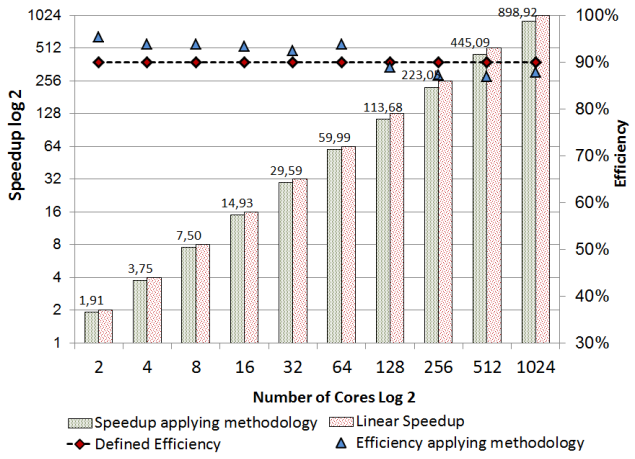
Fig. 6: ZSC-2D-STD-MPI strong scalability analysis.

strong scalability under a defined efficiency.

As demonstrated in this first validation, the maximum strong scalability under a defined efficiency is found near to the value calculated with our method. Our method calculates the ST and from this ST size estimates the ideal number of cores need to achieve the combination of both metrics.

On the other hand, the next validation is focused on demonstrating how to maintain an isoefficiency system, where we can increase the number of processing elements (cores) and problem size while the execution time is maintained approximately constant (weak scalability). In this sense, we start of the ST definition and we increase the problem size using as a reference the supertile size and the number of cores. In a similar manner, we have considered the initial value for the LL-2D-STD-MPI application (table 1). In this case, the ideal value of the supertile is 318 squared. Therefore, if we wish to obtain a constant execution time when we increase the number of cores and the problem size, we have to create the same amount of ST that execution cores in order to assign one ST per core and also, each ST has maintain the same size calculated through the model with the aim of maintaining the overlapping strategy and thus, we can maintain the performance of the system.

Then, the figure 7 shows an example of increasing the problem size and the number of cores proportionally to the ST size and the results illustrate how the execution time is approximately maintained with a small error for the worst case of 5%. This clearly is demonstrated with the values of table inside figure 7 where the problem size is increased and the execution time is approximately constant and the application efficiency is maintained around the threshold value. In this case the efficiency ranges between (+/-) 4% of the efficiency threshold defined, these values depend on the



Cores	Coord(X,Y)		Problem Size (X,Y)		Execution Time	Efficiency	*Speedup*
2	2	1	636	318	3.4042	95.28%	1.91
4	2	2	636	636	3.4562	93.85%	3.75
8	4	2	1272	636	3.4602	93.74%	7.50
16	4	4	1272	1272	3.4762	93.31%	14.93
32	8	4	2544	1272	3.5082	92.46%	29.59
64	8	8	2544	2544	3.4602	93.74%	59.99
128	16	8	5088	2544	3.6522	88.81%	113.68
256	16	16	5088	5088	3.7228	87.13%	223.05
512	32	16	10176	5088	3.7311	86.93%	445.09
1024	32	32	10176	10176	3.6949	87.79%	898.92

Fig. 7: LL-2D-STD-MPI weak scalability analysis.

number of cores and the problem size used.

This last analysis allows us to validate that our method can maintain an execution time constant while the ST is maintained. This phenomenon is motivated that the ST is created under the consideration of the overlapping strategy and the main objective of this strategy is to hide the effects of communications and therefore increases the performance of the application. This method is possible to be applied due to the deterministic behavior of the SPMD applications.

6. Conclusion and Future works

This work has described an efficient method to improve the strong and weak scalability under a defined efficiency of SPMD applications executed on multicore clusters. In this sense, we have used a methodology which defines four execution phases (characterization, tile distribution model, mapping and scheduling) and allows us to determine the ideal number of cores and the supertile size need to maintain a relationship between efficiency and speedup. The examples described in the performance evaluation have demonstrated that using the supertile size and the number of core obtained through the analytical model, we can obtain the maximum strong scalability point with a small error which can vary between (+/-4%) depending of the SPMD application tested and the multicore system used.

Moreover, the weak scalability analysis has demonstrated that if we increase the problem size according to the relationship of the supertile size, we can maintain the efficiency and speedup conditions. In this sense, the weak scalability allows us to determine the ideal problem size for a specific number of cores, that is, our method seeks that the application can be adapted as best possible to the parallel environment with the aim of obtaining a linear scalability. Hence, maintaining the condition of the overlapping between the edge communication and internal computation of the ST, we can obtain an execution with a small error, as was evidenced in the experimentation. The future work is focused on applying our method in hybrid parallel execution (Multicore and GPU), where another communication level is included.

References

[1] I. M. Nielsen and C. L. Janssen, "Multicore challenges and benefits for high performance scientific computing," *Scientific Programming*, vol. 16, pp. 277–285, 2008.

[2] A. Kayi, T. A. El-Ghazawi, and G. B. Newby, "Performance issues in emerging homogeneous multi-core architectures." *Simulation Modelling Practice and Theory*, pp. 1485–1499, 2009.

[3] G. Cong and D. A. Bader, "Techniques for designing efficient parallel graph algorithms for smps and multicore processors," *The Fifth International Symposium on Parallel and Distributed Processing and Applications (ISPA07)*, pp. 137–147, 2007.

[4] R. Buyya, *High Performance Cluster Computing: Architectures and Systems, Vol. 1*. Prentice Hall , New Jersey USA, 1999.

[5] R. Muresano, D. Rexachs, and E. Luque., "Methodology for efficient execution of spmd applications on multicore clusters," *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID), IEEE Computer Society*, pp. 185–195, 2010.

[6] L. M. Liebrock and S. P. Goudy, "Methodology for modelling spmd hybrid parallel computation," *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 8, pp. 903–940, 2008.

[7] G. Mercier and J. Clet-Ortega, "Towards an efficient process placement policy for mpi applications in multicore environments," *EuroPVM/MPI 2009*, pp. 104–115, 2009.

[8] K. Vikram and V. Vasudevan, "Mapping data-parallel tasks onto partially reconfigurable hybrid processor architectures," *Trans. on Very Large Scale Integration Systems*, vol. 14, no. 9, p. 1010, 2006.

[9] A. L. Olivier Beaumont and Y. Robert, "Data allocation strategies for dense linear algebra on two-dimensional grids with heterogeneus communication links," Institut Natio. de Recherche en informatique et en automatique, Tech. Rep., 2001.

[10] F. Trahay, E. Brunet, A. Denis, and R. Namyst, "A multithreaded communication engine for multicore architectures," *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–7, 2008.

[11] L. C. Pinto, L. H. B. Tomazella, and M. A. R. Dantas, "An experimental study on how to build efficient multi-core clusters for high performance computing," *11th IEEE International Conference on Computational Science and Engineering*, pp. 33–40, 2008.

[12] V. der Wijngaart and H. Jin, "Nas parallel benchmarks, multi-zone versions," NASA Advanced Supercomputing Division Ames Research Center, USA, 94035-1000, Tech. Rep., 2003.

[13] A. Hoisie, O. Lubeck, and H. Wasserman, "Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications," *International Journal of High Performance Computing Applications*, vol. 14, pp. 330–346, 2000.

[14] L. Pastor and J. L. Bosque, "An efficiency and scalability model for heterogeneous clusters." *CLUSTER '01: Proceedings of the 3rd IEEE International Conference on Cluster Computing*, p. 427, 2001.

[15] A. Grama, "Isoefficiency: measuring the scalability of parallel algorithms and architectures," *IEEE, Parallel & Distributed Technology: Systems & Applications*, vol. 3, pp. 12 – 21, 1993.

[16] C. Bekas, A. Curioni, P. Arbenz, C. Flaig, G. H. van Lenthe, R. Müller, and A. J. Wirth, "Extreme scalability challenges in micro-finite element simulations of human bone," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 16, pp. 2282–2296, 2010.

[17] R. Muresano, D. Rexachs, and E. Luque., "Combining scalability and efficiency for spmd applications on multicore clusters," *The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications.*, pp. 638–644, 2011.

Performance Comparison Between Cg-based and CUDA-based Matrix Multiplications

Luke West† **and Jong Kwan Lee**
Dept. of Computer Science, Bowling Green State Univ., Bowling Green, OH 43403, U.S.A.

Abstract— *In this paper, we compare the performances of Cg-based and CUDA-based GPU programming APIs. In particular, their performances on squared matrix multiplications are considered. We also discuss other aspects of these widely-used GPU programming APIs. This work can help gain insight on various applications that involve matrix multiplication that are better suited for a specific GPU programming API.*

Keywords: GPU Computing, Cg Programming, CUDA Programming, Matrix Multiplication

1. Introduction

While microprocessors based on a single CPU drove rapid performance increases and cost savings in computing applications for many decades, programmable graphics processing units (GPUs), driven by the insatiable market demand for real time and high-definition 3D graphics, have been the primary high performance parallel computing tool in recent years; GPUs have become extremely powerful parallel computing units with multi-threaded many-core processors that can process not only graphical operations, but also other general-purpose operations.

Utilization of such GPUs has been considered in many research areas. For instance, there have been many attempts to apply GPU-based processing in visualization. Some of these studies involved the development of fast and accurate rendering algorithms using volumetric datasets (e.g., [16], [15]). GPU-based processing has also been applied to computer vision (e.g., [6]). Scientific simulation is another area where GPU processing is often considered (e.g., [14], [12]). Many of these research efforts and applications benefit from exploiting data parallelism on GPUs.

In this paper, we compare the performance of two widely-used GPU programming models–Cg (C for graphics) [8] and CUDA (Compute Unified Device Architecture) [9]– for squared matrix multiplication. (Other GPU programming models include OpenGL Shading Language (GLSL) and High Level Shading Language (HLSL) for DirectX.) This work can help gain insight on various applications involving matrix multiplications that are better suited for a specific GPU programming API. We are unaware of any prior reports on Cg and CUDA performance comparison.

Next, some background information on Cg and CUDA is discussed.

Since 2002, the arrival of the fourth and current generation of GPUs, GPUs supported programmable vertex and fragment shaders. GPU shaders denote the GPU's streaming processing units that perform predefined graphical operations, as well as user-defined operations (including non-graphical operations), in parallel without directly exposing their parallel processing operations to the programmer. When utilizing GPUs for general-purpose applications, tasks that are typically processed on a CPU can be mapped to a GPU and *processed along the GPU pipeline*. A GPU pipeline is the sequence of graphical processing stages that operate in parallel and in a fixed order on a GPU. A typical GPU pipeline is shown in Figure 1 (i.e., the five GPU stages are inside the red box). Here, we note that in applications employing programmable GPUs, fragment shaders are typi-

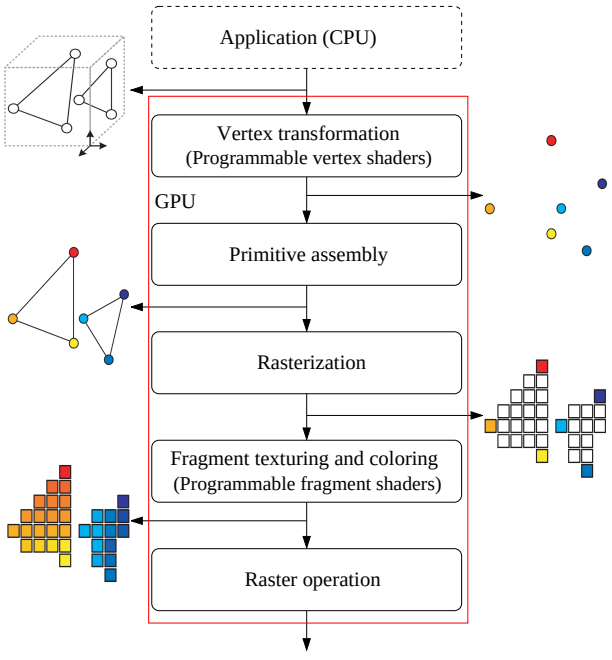


Fig. 1: GPU Pipeline (adapted from [3], [8]) considered for Cg API

†Corresponding Author, Email: lukew@bgsu.edu

cally relied upon more heavily than vertex shaders. One main reason is that there are more fragment shaders than vertex shaders on a typical programmable GPU. In addition, the memory access on fragment shaders is more straightforward than it is on vertex shaders. (For example, the outputs of the vertex shader must pass through all the other pipeline stages before accessing the memory [11]; thus, direct memory access is not straightforward on the vertex shaders.)

Cg is one widely-used GPU programming API that enables programming of the type of GPU discussed above. It is a modified version of the C programming language that includes some new data types that create a more suitable programming environment when utilizing GPUs through the GPU pipeline. Here, we note that some of the later programmable GPUs (e.g., Nvidia GeForce 8 Series and later) have an additional programmable logic unit known as the geometry shader. Geometry shaders are used after the vertex transformation stage and before the primitive assembly stage in the GPU pipeline. This enables generation and elimination of geometry primitives on the fly on the GPUs. However, since the Nvidia GeForce 8 Series GPUs, the concept of shaders has evolved into multi-threaded, unified GPU processors.

While many computing applications have used Cg in GPU computing, this trend changed when CUDA [9] was released in 2007. CUDA provides a new general-purpose parallel programming interface for the GeForce 8 Series (G80) and its successor GPUs. In particular, CUDA-based programs no longer run through the GPU pipeline (i.e., there is no need to perform general-purpose computations within the graphics pipeline); thus, programmers do not need to learn the graphics pipeline to utilize GPUs in their applications.

Figure 2 (a) shows a CUDA-enabled GPU architecture. As shown in the figure, it is organized into an array of highly threaded streaming multiprocessors (represented by the green cells). CUDA-based programs utilize these multiprocessors to exploit data parallelism for extremely high parallel computing performance. In particular, these multiprocessors available on GPU can create and execute thousands of threads simultaneously. For example, the GeForce 200 series GPU can support up to 1024 threads per processor, which sums up to about 30,000 threads in total. We note that CUDA-enabled GPUs also provide a very high memory bandwidth (e.g., G80 supports 86.4 GB/s of memory bandwidth). One key advantage of CUDA over Cg is its ability to use GPU's shared memory for much faster data access. (Modern GPUs have a large amount of global memory and a relatively small amount of shared memory. The global memory is slow whereas the shared memory is fast.) In Figure 2 (b), the latest CUDA compute architecture, the FERMI [10], is shown. FERMI architecture provides not only an increased number of multiprocessors, but also improved performance for double precision computations, true cache hierarchy, more shared memory, faster context switching, and faster atomic operations.

Fig. 2: CUDA Architectures: (a) CUDA-enabled GPU Architecture (G80 GPUs, borrowed from [5]) and (b) Latest CUDA Compute Architecture (FERMI, borrowed from [10])

2. Related Work

Next, some prior work in matrix multiplication is briefly discussed.

The complexity of the standard matrix multiplication using three nested loops is $O(n^3)$. Strassen [1] first reported that the standard matrix multiplication was not the optimal solution. The report presented another way to perform matrix multiplication (i.e., the Strassen's algorithm [1]) that recursively partitioned the matrices into smaller blocks and then performed multiplications. This recursive algorithm's complexity is $O(n^{2.81})$. Another fast matrix multiplication algorithm is the block matrix multiplication algorithm [13] which utilizes memory coherency in matrix multiplication.

There have also been GPU-based matrix multiplication studies. Some of these were based on Cg (e.g., [4], [2]) and some recent studies were based on CUDA (e.g., [17], [7]). However, to our knowledge, there has not been any report that presented a performance comparison of Cg-based and CUDA-based matrix multiplication.

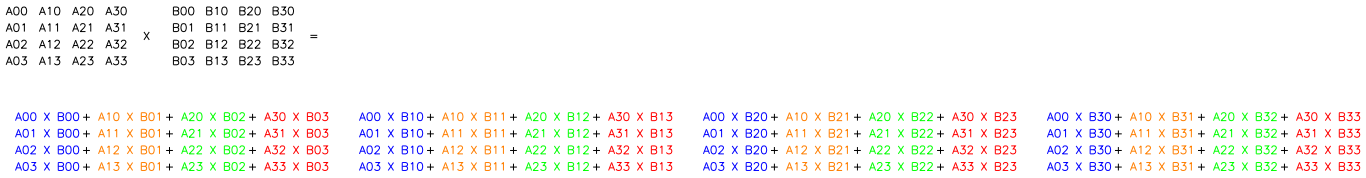


Fig. 3: Illustration of Two 4×4 Cg-based Matrix Multiplication

3. GPU-based Matrix Multiplication

In this section, the Cg-based and CUDA-based squared matrix multiplications used in this paper are described.

Our Cg-based matrix multiplication utilizes the fragment shaders by using textures on GPUs. (A texture on the GPU can be considered similar to an array on the CPU [11].) Specifically, each matrix on the CPU is mapped to a GPU texture. Then, using a simple querying of texture elements, the Cg program iteratively performs the matrix multiplication. In particular, each iteration takes one set of matrix elements from each of the two input matrices, computes the product of the values, and stores the result. Figure 3 illustrates this iterative process of matrix multiplication for two 4×4 matrices. In the figure, the elements of both input matrices (i.e., A and B) are denoted as AXY and BXY, where X and Y denote the column and row indices, respectively. In the Cg program, the multiplication operations are performed in the order of blue, orange, green, and red element multiplications in each iteration. The GPU's fragment shaders perform the multiplications in parallel. The Cg-based code is shown in Figure 4.

The CUDA-based matrix multiplication we consider here is based on the CUDA-based matrix multiplication in [5]. Similar to Cg-based matrix multiplication, input matrices are mapped to the GPU's memory. Then, each thread (among thousands GPU threads) computes one element in the product matrix. For very large matrices, the *tiling* is employed. Tiling divides the matrices into smaller sub-matrices and performs the matrix multiplication. In addition, for faster memory referencing, the GPU's shared memory is also used.

```
void _cg_mat_mul( in float2 coords: WPOS,
                  uniform samplerRECT texA,
                  uniform samplerRECT texB,
                  uniform float mSize,
                  out float matC : COLOR0 )
{
    float matA, matB;
    float2 ACoords, BCoords;

    for(float i=0; i<mSize; i++)
    {
       ACoords = float2(i+0.5, coords.y);
       matA = texRECT(texA, ACoords);

       BCoords = float2(coords.x, i+0.5);
       matB = texRECT(texB, BCoords);

       matC = matC + matA*matB;
    }
}
```

Fig. 4: Cg-based Matrix Multiplication Code

4. Experimental Results

We compared the processing times of the Cg-based and CUDA-based squared matrix multiplications using six different matrix sizes (i.e., 256^2, 512^2, 1024^2, 2048^2, 4096^2, and 8192^2) on two different PCs (running Linux) that have different GPUs. Trimmed average timings were used for matrix multiplication processing time measurements.

The first (Type 1) PC has a 2.8 GHz Intel Core i7 CPU with 12 GB of RAM and dual 1.8 GB Nvidia GeForce GTX 285 GPUs. The GTX 285 GPU has 240 CUDA processors and supports a memory bandwidth of 159 GB per second. The second (Type 2) PC has a 3.4 GHz Intel Core i7 CPU with 16 GB of RAM and one 1.5 GB Nvidia GeForce GTX 480 GPU. The GTX 480 GPU has 480 CUDA processors and supports a memory bandwidth of 177.4 GB per second. We note that only GTX 480 GPU is a FERMI GPU.

Table 1 shows the processing times of CPU-based, Cg-based, and CUDA-based squared matrix multiplications on Type 1 PC (with dual GeForce GTX 285 GPUs). Here, the block matrix multiplication algorithm [13] was used for the CPU-based matrix multiplication. As shown in the table, the Cg-based and CUDA-based matrix multiplications executed much faster than the CPU-based version. In addition, the CUDA-based version was faster than the Cg-based version.

In Table 2, the GPU-based matrix multiplications on Type 2 PC (with a Geforce GTX 480 GPU). As shown in the table, the CUDA-based version executed faster than the Cg-based version.

Figure 5 summarizes our performance comparisons. In Figure 5 (a), the speedups of the Cg-based and CUDA-based matrix multiplications over the CPU-based matrix multiplication is shown. As shown in the table, while the Cg-based version was about 10 times faster for the 256^2 case and about 340 times faster for the 8192^2 case, the performance of CUDA-based version was better (e.g., 100 times faster for the 256^2 case and about 500 times faster for the 8192^2 case). In Figure 5 (b), the performance comparisons of Cg and CUDA (i.e., Cg processing times over CUDA programming times) on two different GPUs are shown. As shown in

Table 1: Processing Times (in seconds) on Type 1 PC (dual GTX 285 GPUs): CPU-based, Cg-based, and CUDA-based Squared Matrix Multiplications

Matrix Size	CPU	Cg	CUDA
256^2	0.03	0.003	0.0003
512^2	0.31	0.015	0.002
1024^2	3.19	0.094	0.015
2048^2	25.56	0.668	0.129
4096^2	205.5	5.182	1.032
8192^2	4146.4	12.103	8.276

Table 2: Processing Times (in seconds) on Type 2 PC (GTX 480 GPU): Cg-based and CUDA-based Squared Matrix Multiplications

Matrix Size	Cg	CUDA
256^2	0.004	0.0002
512^2	0.013	0.001
1024^2	0.065	0.009
2048^2	0.448	0.073
4096^2	3.397	0.582
8192^2	9.334	4.852

the table, the CUDA-based version executed faster than the Cg-based version on both GPUs. Specifically, the CUDA-based version was about 10.0, 7.5, 6.3, 5.2, 5.0, and 1.5 times faster using the GTX 285 GPUs and about 20.0, 13.0, 7.2, 6.1, 5.8, and 1.9 times faster on the GTX 480 GPU, for 256^2, 512^2, 1024^2, 2048^2, 4096^2, and 8192^2 matrix multiplications, respectively. Use of the shared memory in CUDA-based version was one of the key factors for this better performance.

5. Conclusion and Discussion

In this paper, we considered two of the most widely-used GPU computing APIs (i.e., Cg and CUDA) and compared their performances for squared matrix multiplication. To our knowledge, this work is the first report to compare the Cg and CUDA performance. A simple Cg-based matrix multiplication code is also presented.

For the matrix multiplications, while both Cg and CUDA outperformed the CPU-based processing, CUDA-based was 1.5–20.0 times faster than Cg-based version. The relative CUDA performance was better for smaller matrix sizes than for the large matrix sizes. For example, the CUDA-based version was 10–20 times faster for 256^2 matrices while it was only 1.5–1.9 times faster for 8192^2 matrices. Here, we note again that the CUDA-based version utilized the GPU's shared memory. CUDA-based version without using the shared memory was about 3.5–6.6 times slower than CUDA-based version with the shared memory. For large matrix sizes

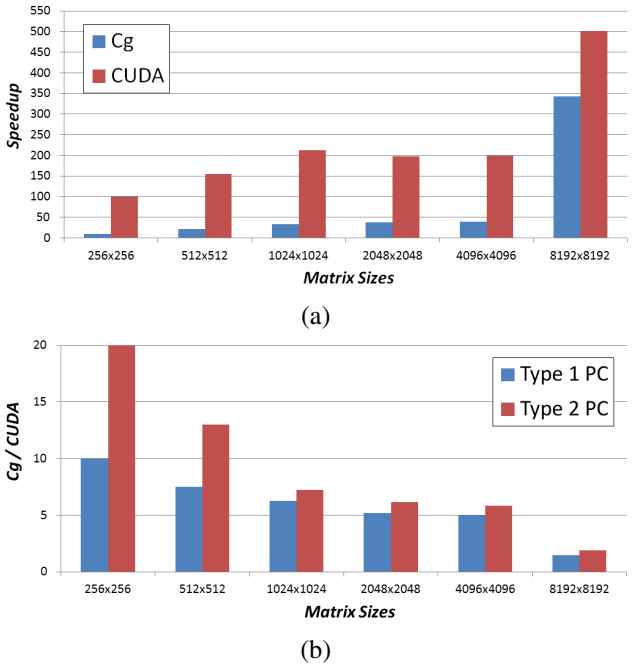


Fig. 5: Performance Comparisons: (a) Speedups of GPU-based versions (vs. CPU-based version) and (b) Cg and CUDA Performance Comparisons (Cg processing time/CUDA processing time) on Type 1 PC (dual GTX 285 GPUs) and on Type 2 PC (GTX 480 GPU)

(e.g., 4096^2 and 8192^2), the CUDA-based version without using shared memory was even slower than the Cg-based version. Shared memory usage is not supported in Cg-based programs.

For the programmers who are familiar with the GPU pipeline, the Cg API is easy to understand. For matrix multiplication, Cg-based code was shorter than that of the CUDA-based code. Here, we note that using Cg-based matrix multiplication for very large matrices, tiling is also needed since there is a limit on the GPU texture size on GPUs. For programmers who are not familiar with the GPU pipeline, it is difficult to write Cg programs. Programmers with some parallel processing experience find it easier to write high performance programs using the CUDA API than the Cg API.

We have also found that debugging is much easier using the CUDA-based API. There is no debugging tool support provided for the Cg API. The programmers have to transfer values through the GPU pipeline to the CPU, then display and check the values manually using Cg API. For the CUDA-based API, the programs can be executed in device emulation mode. In this mode, threads are executed sequentially on the CPU and other debugging tools (e.g., break points) are also supported.

These different aspects of Cg and CUDA APIs reported

in this paper can assist scientists in choosing which GPU programming API to use for their high performance applications.

Acknowledgment

This work was partially supported by NVIDIA's Professor Partnership Program.

References

[1] V. Strassen, "Gaussian Elimination is not Optimal,", *Numerische Mathematic*, Vol. 13 (4), pp. 354–356, 1969.

[2] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication," *Proc., Graphics Hardware*, pp. 133–137, Grenoble, France, August 29–30, 2004.

[3] N. Goodnight, R. Wang, and G. Humphreys, "Computation on Programmable Graphics Hardware," *IEEE Computer Graphics and Applications*, Vol. 25 (5), pp. 12–15, 2005.

[4] J. D. Hall, N. A. Carr, and J. C. Hart, "Cache and Bandwidth Aware Matrix Multiplication on the GPU," University of Illinois at Urbana-Champaign Technical Report, UIUCDCS-R-2003-2328, 2003.

[5] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*, Morgan Kaufmann, 2010.

[6] J. K. Lee, B. A. Wood, and T. S. Newman, "Very Fast Ellipse Detection using GPU-based RHT," *Proc., 19th Int'l Conf. on Pattern Recognition*, pp. 1–4, Tampa, Florida, December 8–11, 2008.

[7] J. Li, S. Ranka, and S. Sahni, "Strassen's Matrix Multiplication on GPUs," *Proc., IEEE 17th Int'l Conf. on Parallel and Distributed Systems*, pp. 157–164, Tainan, Taiwan, December 7–9, 2011.

[8] W. R. Mark, R. Steven, G. Kurt, A. Mark, and J. Kilgard, "Cg: A System for Programming Graphics Hardware in a C-like Language," *ACM Transactions on Graphics*, Vol. 22, pp. 896–907, 2003.

[9] NVIDIA, CUDA, http://www.nvidia.com/object /cuda_home_new.html, 2007.

[10] NVIDIA, FERMI, http://www.nvidia.com/conte nt/PDF/fermi_white _papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2010.

[11] M. Pharr and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison Wesley, 2005.

[12] D. P. Playne and K. A. Hawick, "Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA," *Proc., The 2009 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '09)*, pp. 104–110, Las Vegas, July 13–16, 2009.

[13] R. Schreiber, "Block Algorithms for Parallel Machines," *Numerical Algorithms for Modern Parallel Computer Architectures*, Springer-Verlag, pp. 197–208, 1988.

[14] R. Strzodka, M. Doggett, and A. Kolb, "Scientific Computation for Simulations on Programmable Graphics Hardware," *Simulation Modelling Practice and Theory, Special Issue: Programmable Graphics Hardware*, Vol. 13 (8), pp. 667–680, 2005.

[15] N. Tatarchuk, J. Shopf, and C. DeCoro, "Advanced Interactive Medical Visualization on the GPU," *Journal of Parallel and Distributed Computing*, Vol. 68 (10), pp. 1319–1328, 2008.

[16] Q. Zhang, R. Eagleson, and T. M. Peters, "Graphics Hardware Based Volumetric Medical Dataset Visualization and Classification," *Proc., Medical Imaging 2006: Visualization, Image-Guided Procedures, and Display (SPIE Vol. 6141)*, pp. 61412T, March, 2006.

[17] Y. Zhang and J. D. Owens, "A Quantitative Performance Analysis Model for GPU Architectures," *Proc., 17th IEEE Int'l Symposium on High-Performance Computer Architecture*, pp. 382–393, San Antonio, Texas, February 12–16, 2011.

Parallel Benefit on Different Programming Paradigms

Chau-Yi Chou, Sheng-Hsiu Kuo, Chih-Wei Hsieh, Tsung-Che Tsai and Hsi-Ya Chang

National Center for High-Performance Computing, Taiwan

Abstract - *Multi-core platforms become ubiquitous nowadays. Even laptops contain multi-core processors now. There are multiple cores in a chip or socket or die. A computing node contains multiple chips. Multi-core platforms are rapidly increasing and the number of cores on these platforms is increasing rapidly too. How to enjoy the benefits of parallel computing on the multi-core platforms plays a key role in High Performance Computing. With the increasing complexity of modern multi-core processors, the problem of distributing a software application across different cores to maximize the utilization of the computing power becomes more and more difficult. Different programming patterns great influence the program performance. We implement different parallel programming paradigms on Himeno Benchmark via hybrid MPI/OpenMP in this paper. Moreover, we will evaluate the performance of those on NCHC GPU Cluster and NCHC ALPS. We establish a Roofline Model for NVIDIA GT200, too. We hope the results can give some useful information to the user of HPC.*

Keywords: Performance Evaluation, Parallel Programming, MPI, OpenMP

1 Introduction

Multi-core platforms become ubiquitous nowadays. Even laptops contain multi-core processors now. There are multiple cores in a chip or socket or die. A computing node contains multiple chips. For example, Intel X5472 consists of dual die quad-core CPUs manufactured on a 45 nm process. Multi-core platforms are rapidly increasing and the number of cores on these platforms. Moreover, many users of High Performance Computing, HPC, adopt the multi-core platforms. How to enjoy the benefits of parallel computing on the multi-core platforms plays a key role in HPC.

With the increasing complexity of modern multi-core processors, the problem of distributing a software application across different cores to maximize the utilization of the computing power becomes more and more difficult. Different programming patterns big influence the program performance. Two common programming patterns of parallel program are Message Passing Interface [1], MPI, and OpenMP [2]. MPI had widely used to parallel program on traditional parallel platforms and PC Cluster since 1994. Scientists have obtained a great help on MPI programming pattern in the last few decades. Even it enters multi-core platforms ear. Naturally it is in the wake of overheads, such as message passing inside a node and duplicate memory location.

OpenMP programming pattern is implemented for high efficient computing on Symmetrical multiprocessor system, especially on a multi-core platform. It adopts a fork-and-join execution model, that is, it is thread level parallelism. The behavior of compute is in system bus level and threads are able to share a memory space, but it is limited by scale. The version 2.5 of standard was released in 2005. Most of the compilers (Fortran or C or C++) support the functions of "directive", runtime libraries, and environment variables. Moreover, the version 3.0 of standard, which contains "task" parallelism implementations, was released in May 2008.

We intuitively think the hybrid MPI/OpenMP computing that MPI mainly handles inter message passing, while OpenMP focus on intra computing. But it inherits diverse hardware characteristics, such as the number of multi cores in a die, the number of die in a node, the bandwidth of memory and system bus, how many cores shared resources, and so on. In general, there are two hybrid programming paradigms on multi-core platforms. One is like this model, while the other model is that one adopts a "Parallel Region" directive of OpenMP and master thread to handle the message passing.

We adopt Himeno Benchmark [3], which performs computations of 19-points stencil, to evaluate the performance on different programming patterns of parallel program on NCHC (National Center for High-Performance Computing) platforms, "GPU Cluster", which contains 16 of Intel X5472, and NCHC ALPS hereafter. [4] depicts that the "Stencil" computation, such as Himeno Benchmark, is limited by the bandwidth of memory based on Roofline Model. It is conceivably improved the performance of parallel programs based on Thread Level Parallelization only.

Firstly, we evaluate the performance of Himeno Benchmark on 8 cores in a node on GPU Cluster on different compilers. We implement the hybrid MPI/OpenMP programming patterns on Himeno Benchmark and come out these results on hybrid mode and pure MPI mode on different mapping patterns of 8 nodes (64 Cores) of GPU Cluster. We will present the results of Himeno Benchmark and High Performance Linpack, HPL[5], for CPU binding on NCHC ALPS. We also illustrate the Roofline model on NVIDIA GT200. We hope the results can give some useful information to the user of HPC.

2 Test beds and software

We adopt two test beds, NCHC GPU cluster built in 2010 and upgraded in 2011 and NCHC ALPS built in 2011. We implement MPI/OpenMP hybrid for Himeno Benchmark and evaluate the performance on both platforms. HPL is employed for CPU binding on NCHC ALPS.

2.1 NCHC GPU Cluster

GPU Cluster contains 16 of Intel X5472, which consists of dual die quad-core CPUs manufactured on a 45 nm process. The motherboard adopts Tempest i5400XT (S5396) and Intel 5400 Chipset as shown in Figure 1. Figure 2 shows the logic picture, system bus and bandwidth of memory.

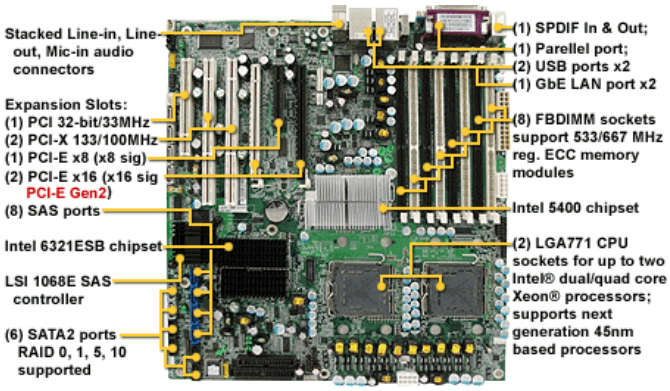


Figure 1. Motherboard of GPU Cluster

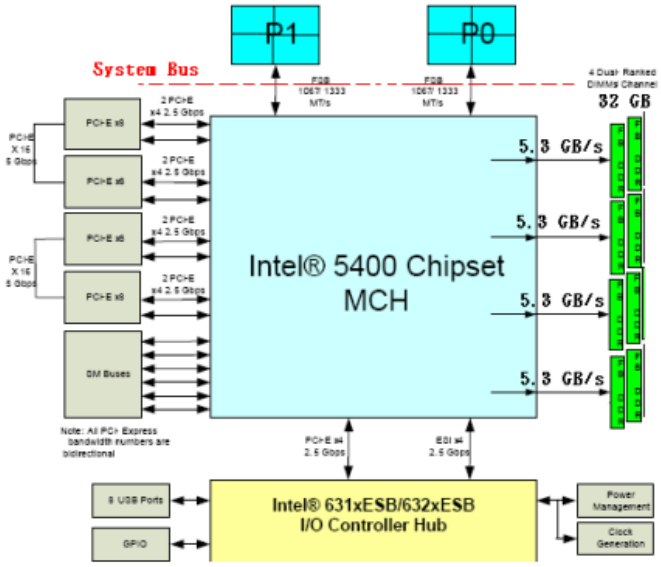


Figure 2. Intel 5400 Chipset

A computing node contains 32 GB RAM and a DDR Infiniband connected together. We adopt OpenSuSE 11.0 as Operation System, OS, different compilers, its version and its compiler option as Intel 11.0 (**`ifort -openmp -O3 -fast`**), PGI 9.0 (**`pgfortran -mp -O3 -fast`**), and GNU 4.3.2 (**`gfortran -fopenmp -O3`**), and MPI middleware as OpenMPI 1.2.8.

2.2 NCHC ALPS

The hardware of computing nodes on NCHC ALPS consists of 600 of Acer AR585 as shown in Figure 3. They are connected together with Qlogic InfiniBand in 4x QDR (40Gb) and the bandwidth throughput of this system achieves 51.8 Tbps. In logic point of view, the system comprises 8 computing clusters, which consists of 4 of AMD Opteron 6174 inside 12 cores running at 2.2 GHz, that is, 48 cores a node sharing 128 GB RAM in 4-memory-controller non-uniform memory access architecture, and 1 large memory cluster that includes 4 of AMD 6136, which comprises 8 core running at 2.2 GHz, that is, 32 cores a node sharing 256GB RAM. There are 25,600 computing cores of AMD Opteron 6100 in this system and the maximal Linpack achieves at 177 TFlops and at 42th place in Top500 list in June 2011 [6].

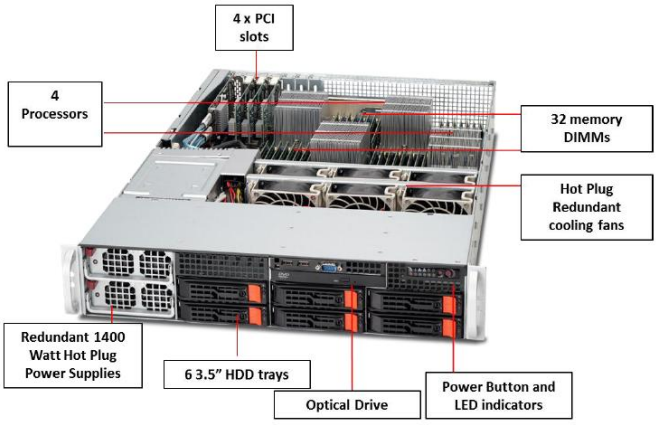


Figure 3. The architecture of Acer AR585

This system adopts the Novell SuSe Linux Enterprise 11 SP 1 for its operation system. The parallel file system is Lustre. Platform LSF is for job scheduler and queuing system. Message passing interface libraries are installed such as Platform MPI, OpenMPI, mvapich, and so on. The debug tool is Allinea DDT, Distribut Debugging Tool. There are four compilers in this system: Intel, PGI, Open64, and GNU. Intel MKL and AMD ACML, AMD Core Math Library, inherit this system for math libraries.

2.3 Himeno Benchmark

To solve the pressure commonly adopts Poisson equation solver in computational fluid dynamic, such as incompressible Navier-Stokes equations solver. The Poisson equation is shown as Figure 4 and the kernel computing of Himeno Benchmark. Use central finite difference and Jacobi iteration. To calculate the value of a point requires reading 18 points of neighbors, named it as "19-Points Stencil". It performs 34 of floating-point operations and uses 14 of three-dimensional matrices per iteration. The computational intensity achieves 0.6 *Flop/Byte* on a problem size of 1024×512×512 in single precision compute (required around 14*GB*) sequentially.

Poisson Equation: $\Delta p = \rho$

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} + \alpha \frac{\partial^2 p}{\partial x \partial y} + \beta \frac{\partial^2 p}{\partial x \partial z} + \gamma \frac{\partial^2 p}{\partial y \partial z} = \rho$$

$$\frac{p_{i+1,j,k} - 2p_{i,j,k} + p_{i-1,j,k}}{\Delta x^2} + \frac{p_{i,j+1,k} - 2p_{i,j,k} + p_{i,j-1,k}}{\Delta y^2}$$
$$+ \frac{p_{i,j,k+1} - 2p_{i,j,k} + p_{i,j,k-1}}{\Delta z^2}$$
$$+ \alpha \frac{p_{i+1,j+1,k} - p_{i-1,j+1,k} - p_{i+1,j-1,k} + p_{i-1,j-1,k}}{4\Delta x \Delta y}$$
$$+ \beta \frac{p_{i+1,j,k+1} - p_{i-1,j,k+1} - p_{i+1,j-1,k} + p_{i-1,j,k-1}}{4\Delta x \Delta z}$$
$$+ \gamma \frac{p_{i,j+1,k+1} - p_{i,j+1,k-1} - p_{i,j-1,k-1} + p_{i,j-1,k-1}}{4\Delta y \Delta z} = \rho_{i,j,k}$$

Figure 4. Himeno Benchmark

2.4 The High Performance Linpack, HPL

The High Performance Linpack[5], HPL, employs the LU decomposition to solve a dense $N \times N$ system of linear equation in a floating point workload of 2/3 N^3+2N^2. HPL utilizes LU factorization with row partial pivoting to solve a dense linear system while using a two-dimensional block-cyclic data distribution for load balance and scalability.

3 Methodology

We implement two hybrid models, which both models will be described in detail in next Section, for Himeno benchmark in Fortran 90 and evaluate the performance on NCHC GPU Cluster first. The results are performed for GPU version on NCHC GPU Cluster, too. Moreover, we establish a Roofline model for GT200.

Since NCHC ALPS inherits the non-uniform memory access, NUMA, and consists of 48 cores and 8 of NUMA servers as shown in Figure 5, we expect the limit of system bus and memory bandwidth. In order to get the best performance, we adopt the CPU binding for memory use efficiently. First we find the rank pattern via HPL and evaluate the performance. Next, we compare the performance of CPU binding with those for other models via Himeno benchmark. The performances of Himeno benchmark via different compilers are shown, too.

	0	1	2	3	4	5	6	7
0	10	16	16	22	16	22	16	22
1	16	10	22	16	22	16	22	16
2	16	22	10	16	16	22	16	22
3	22	16	16	10	22	16	22	16
4	16	22	16	22	10	16	16	22
5	22	16	22	16	16	10	22	16
6	16	22	16	22	16	22	10	16
7	22	16	22	16	22	16	16	10

Figure 5. Distance of NUMA server

4 Results

We follow 95% confidence to evaluate the results and employ the ganglia and "top" command to monitor the status of the system for dedicated usage.

4.1 NCHC GPU Cluster

We show that the performance of this program at 3.35 *GFlops*, 2.85 *GFlops*, and 2.57 *GFlops* on 8 cores a node via Intel, PGI, and GNU, respectively. It is expectable. Intel compiler gets more benefit of intrinsic computation involving Streaming SIMD Extensions than the others. The performance score of Intel compiler slightly surmounts the peak performance of 3.20 *GFlops*. Commercial PGI compiler shows the performance based on standard computational pattern. The score is lower than the peak performance. The public GNU compiler obtains the worst performance in three compilers.

We implement two hybrid MPI/OpenMP parallel programming patterns: First Model (Hybrid Model 1) is that MPI handles inter message passing between nodes, while OpenMP handles intra computation between 8 cores a node. Its advantage is that the data and program flow are clear. That is, the program is able to be divided into some sub-program blocks, which many sub-program blocks perform heavy operation, while few sub-program blocks perform message passing between nodes. The downside is that all threads are idle on message passing. Amdahl's law points that it big decreases the parallel performance!

The other model (Hybrid Model 2) is that one use a parallel region involves all operations and message passing, which master thread performs message passing between nodes. The race conditions become more and more. The program structure and data flow becomes complicated, too. That is, a programmer requires more human time to better performance. It is opportunity to get parallel efficiency in hidden overhead of message passing carefully. The first model shows 15.0 *GFlops* on 8 nodes with Intel compiler and OpenMPI 1.2.8, while the second model shows 22.50 *GFlops*. It is expectable that the last model presents better performance than those on the first model. We very surprise that different programming patterns can improve 7.5 *Gflops*!

Though MPI 2 has new features of communicator management and OpenMP 3.0 increases the task parallelism, we don't unfortunately perform our overlap version between computation and massage passing (**MPI_THREAD_MULTIPLE**) on our test beds. We adopt MPI 1.2 standard and OpenMP 2.5 to implement our hybrid MPI/OpenMP parallel programming pattern as end-users. The second model of hybrid MPI/OpenMP parallel programming patterns with Intel compiler and OpenMPI 1.2.8 are evaluated on GPU Cluster. Therefore, MPI can perform diverse MPI task mapping, for example a node has 1×1×8, 1×8×1, 8×1×1, 1×2×2, and so on. That is, the data decomposition is 1-dimension in z- or y- or x-direction or 2-dimension in y- and z-direction.

Like MPI mapping, hybrid MPI/OpenMP parallel programming pattern contains large amount of combination of MPI and thread. We evaluate all mapping on combinations of MPI Processes and threads to get interest results. First of all, we define "1×2×4×8" as x-, y-, and z-directions of MPI Processes and the number of threads a MPI Process, that is, x-, y-, and z-directions use 1, 2, 4 MPI Process, respectively, and a MPI Process uses 8 threads. The special case "4×4×4×1" means pure MPI programming paradigm, because of a thread used a MPI process. Table 1 shows the partial (better) results in *GFlops* on 8 nodes (64 cores in total) on different programming paradigms. To our surprise, the pure MPI programming paradigm (4×4×4×1) outperforms!

This is because that the Intel compiler abundantly enjoys the benefit of the hardware, such as SSE intrinsic. Hybrid programming paradigm doesn't have enough room for thread level parallel. Our experience via PGI compiler, pure MPI obtains 19.18 *GFlops* vs. Hybrid model achieves 20.99 *GFlops*. Our opinion is confirmed. The other reason is hardware limitation, such as, obstruction of memory bandwidth. We adopt the Phillips's results presented in Cluster 2009 conference to establish a Roofline Model for GT200 as shown in Fig. 5. The performance of CUDA version of Himeno Benchmark achieves at 767 *GFlops!*

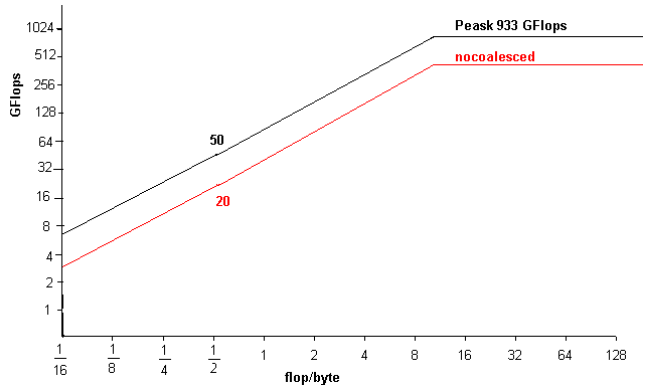


Figure 5. Roofline model on GT200

4.2 NCHC ALPS

For our clear description, we define the notations, **Intel**, **Op64**, **GNU**, and **PGI**, in Table 1. The compiler option is "**Ofast**" for Op64 and GNU, while it is "**fast**" for Intel and PGI.

Table 1. Notations

Notation	Compiler	MPI
Intel	Intel 12.0	openMPI 1.4.3
Op64	Open64 4.2.5	openMPI 1.4.4
GNU	gcc 4.6.2	openMPI 1.4.4
PGI	PGI 11.10	openMPI 1.4.4

We adopt N=1000 for HPL using two cores to obtain the best CPU bind mapping 0-3. We adopt the mapping for **CPU bind** model hereafter and use GNU compiler with AMD acml5.1.0 single thread. Table 2 shows the maximal Linpack in *GFLOPS* for different N=80000 vs. N=100000 with/out CPU bind on 48 core a node. CPU bind model outperforms.

Table 2. Maximal Linpack in *GFLOPS* for different N with/out CPU bind

N	CPU bind	Not CPU bind
80000	286.7	278.0
100000	287.9	274.7

Table 3 depicts the performance of Himeno benchmark in *GFLOPS* for different compilers for $1024 \times 512 \times 512$ problem size in $3 \times 4 \times 4$ partition pattern with/out CPU bind on 48 cores a node. As we expected, CPU bind model outperforms. It is very interesting for CPU bind that PGI outperforms on NCHC ALPS, instead of Intel! It is different from those performed on NCHC GPU Cluster. When we do not use bind processor, Intel outperforms like those on NCHC GPU Cluster. It is because that PGI shows slightly better than Intel for memory affinity on NCHC ALPS.

Table 3. Performance of Himeno benchmark in *GFLOPS* for different compilers for 1024×512×512 problem size in 3×4×4 partition pattern with/out CPU bind

Compiler	CPU bind	Not CPU bind
PGI	36.69	17.02
Intel	35.93	20.12
Op64	31.77	16.22
GNU	20.44	12.12

Table 4 depicts the performance of Himeno benchmark in *GFLOPS* for different partition patterns for 1024×512×512 problem size with/out CPU bind on 48 cores a node. As we expected, CPU bind model outperforms, again. It is very interesting for CPU bind that the three-dimensional partition pattern, $4 \times 4 \times 3$, cannot enjoy the benefits of parallel computing, while the two-dimensional partition pattern, 8×6×1, outperforms. It is different from those on NCHC GPU cluster again.

Table 4. Performance of Himeno benchmark in *GFLOPS* for different partition patterns for 1024×512×512 problem size in with/out CPU bind

partition patterns	CPU bind	Not CPU bind
48×1×1	26.08	9.25
1×48×1	29.25	14.20
1×1×48	27.82	11.53
8×6×1	40.85	26.12
6×8×1	40.24	26.15
6×1×8	36.67	19.54
3×4×4	36.69	17.02
4×3×4	34.79	27.57
4×4×3	40.09	22.30

Table 5 shows the performance of Himeno benchmark in *GFLOPS* for different processor binds for 1024×512×512 problem size in 8×6×1 partition pattern on 48 cores a node.

Rank model, which we define the processor mapping by myself, outperforms. The model that binds each MPI process to a core show comparable results to those on Rank model. The other model, which it binds each MPI process to a processor socket, performs worse results. Every core performs around 60% of workload based on "**top**".

Table 5. Performance of Himeno benchmark in *GFLOPS* for different processor binds for 1024×512×512 problem size in 8 ×6×1 partition pattern on 48 core a node

partition patterns	GFLOPS
Rank model	40.85
Bind each MPI process to a core	40.07
Bind each MPI process to a processor socket	14.75
Not bind processes	26.12

5 Conclusion

We implement two hybrid MPI/OpenMP models for Himeno Benchmark and evaluate the performance on NCHC GPU Cluster and NCHC ALPS. Intel compiler outperforms on NCHC GPU Cluster, while PGI compiler outperforms on NCHC ALPS for CPU bind! It is because that Intel compiler gets more benefit of intrinsic computation involving Streaming SIMD Extensions on Intel platform than PGI and GNU. Moreover, it does not have enough room for thread level parallel via hybrid MPI/OpenMP. Consequently, The pure MPI parallel programming paradigm on 4×4×4 partition pattern outperforms on NCHC GPU Cluster!

Himeno Benchmark and High performance linpack can enjoy the benefits of parallel computing for CPU binding on NCHC ALPS from our various evaluations. The two-dimensional partition pattern, 8×6×1, with PGI compiler outperforms for Himeno Benchmark on NCHC ALPS. Our defined rank model outperforms for Himeno Benchmark and High performance linpack on NCHC ALPS. We establish a Roofline Model for GT200, too.

Acknowledgment

We would like to thank NVIDIA Co. for supporting CUDA version of Himon Benchmark. We are grateful to the National Center for High-Performance Computing for computer time and facilities.

References

[1] Snir, M., Otto, S., Huss-Lederman, S., Walker, D. and Dongarra, J. *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1996

[2] Ayguade, Eduard, Copty, Nawal, Duran, Alejandro, Hoeflinger, Jay, Lin, Yuan, et al. The Design Of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 404-418, 2009

[3] Himeno Benchmark **http://accc.riken.jp/HPC_e/HimenoBMT_e.html**

[4] Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* , volume (52), 65-76, 2009

[5] HPL Web site, **http://www.netlib.org/benchmark/hpl/**.

[6] Top 500 Web site, **http://top500.org**

Evaluation of the 3rd generation Intel Core Processor focusing on HPC applications

Pawel Gepner[1], **David L. Fraser**[1], **and Victor Gamayunov**[2]
[1]Platform and Technology Enabling Group, Intel Corporation, Swindon, UK
[2]Software and Services Group, Intel Corporation, Swindon, UK

Abstract— *In this paper we critically take a look at what the 3rd generation Intel Core processor brings to high performance computing. We compare four generations of Intel CPU based systems and present a performance review of these four platforms. We compare four families of Intel CPU based systems utilizing a single socket platform across a number of HPC benchmarks and micro-benchmark focused on different performance criteria, compare the results and discuss the implications for HPC.*

Keywords: 3rd generation Intel Core i7-3770K processor; benchmarking; multicore; performance evaluation.

1. Introduction

The 3rd generation Intel Core processor family (codename: Ivy Bridge) is tick product in the Intel's "tick-tock" design methodology. This new CPU does not only represent typical shrink to new 22 nm manufacture process but also improve performance of the integrated graphic module as well improving power-management capabilities.

The new graphic unit is completely redefined and provides new level of experience and performance for video and graphics applications. In this paper we will not focus on graphics aspect of performance as we targeted the HPC applications where floating-point performance is the main criteria.

The 3rd generation Intel Core processor family is based on the same micro-architecture philosophy as its predecessor. This is the second generation of CPU products with AVX support and second CPU with integrated graphics processor unit (GPU) in a monolithic chip.

The first members of the 3rd generation Intel Core processor family have been designated for mobile and desktop products and the server version of Ivy Bridge will be introduced afterwards. Power consumption as well as performance per watt characteristic was first design choices for Ivy Bridge. The first priority was to reduce power. After power savings, Intel design team focused more effort on improving the graphics unit.

Although the first members of the 3rd generation Intel Core processor family are focused on the desktop and mobile markets some of the technology implemented is also expected to be very applicable for the HPC segment.

This study compares 4 configurations of systems: one based on the Intel Core 3rd generation processor second on Intel Core 2nd generation processor family the third system is based on Intel Core i7-970 Processor and the fourth one Intel Core i7-870 Processor based system.

The present study uses High Performance Computing Challenge benchmarks, NAS Parallel Benchmarks and micro-benchmark to measure performance at the subsystem level.

To the best of our knowledge, this is first paper to conduct:

- Analytical and extensive performance evaluation and characterization of a system based on the Intel Core 3rd generation processor, using HPCC and NPB suite.
- Complete comparison of Intel Core 3rd generation processor based system with a system based on Intel Core 2nd generation processor family and another systems based on Intel Core i7-970 Processor (codename: Gulftown) and one Intel Core i7-870 Processor (codename: Lynnfield).
- Performance evaluation of Turbo Mode implemented in the Intel Core 3rd generation processor.

To answer the research questions, we have organized the paper in the following way. Section 2 presents architecture and system configuration for all platforms and deeply describes the microarchitecture of Intel Core 3rd generation processor. In section 3 we discuss system performance and analyze results from running benchmarks. Section 4 contains a summary and conclusions of the study.

2. Architecture and Configuration of the Systems

In our study the 3rd generation Intel Core i7-3770K processor is running on new Intel DX79SI desktop board based on Intel X79 chipset with 8GB of system memory (4x2GB DDR3-1600MHz) with Optimal memory configuration. The 2nd generation Intel Core i7-2600 processor based system utilizes the Intel desktop board DP67BA with 8GB system memory (4x2GB DDR3-1333MHz) populating all available DIMM slots. The six cores based system utilizing Intel Core i7-970 processor is based on Intel desktop board DX58SO with 6GB system memory. This platform has 3 memory channels and we utilized all 3 channels with 2GB (1 DIMM 2GB per channel, DDR3-1066MHz). The quad core system

based on Intel Core i7-870 processor uses Intel desktop board DP55WB with 8GB (4x2GB 2 DIMMs per channel DDR3-1333MHz).

All platforms have deployed enterprise Intel SSD X25-E hard disk drive. Operating system installed is in all the platforms the same: RedHat Enterprise Linux 6 kernel ver. 2.6.32-71.el6.x86_64. In all platforms the new version of the Intel software tool kit has also been installed. The Intel Composer XE 2011 includes Intel Compiler 12.0.0.084 as well as the Intel Math Kernel Library (MKL) 10.3. The new compiler and libraries offer advanced vectorization support, including support for Intel AVX and include Intel Parallel Building Blocks (PBB), OpenMP, High-Performance Parallel Optimizer (HPO), Interprocedural Optimization (IPO) and Profile-Guided Optimization (PGO). All performance tools and libraries provide optimized parallel functions and data processing routines for high-performance applications and in addition contain several enhancements, including improved Intel AVX support.

2.1 CPU characteristic

The 3rd generation Intel Core processor family has been manufactured on 22nm Hi-k metal gate silicon technology and has 1.48 billion transistors for quad-core version. This is the increase from 995 million transistors implemented on Sandy Bridge. Die size of new Ivy Bridge is 23% smaller than quad core version of Sandy Bridge. The 3rd generation Intel Core i7-3770K processor die size has an area of 170mm2 and the 2nd generation Intel Core i7-2600 processor has a die size roughly 220mm2. Majority of this new 500 millions of transistors are taken by new graphics unit as CPU‘s part of the chip is almost unchanged from that implemented in Sandy Bride.

Nevertheless same modification has been made on to the cache and memory interface such as memory hashing and reuse of cache line.

The Ivy Bridge has also new next-page prefetcher able to fetch cache lines ahead of use it on.

The new floating-point data type has been also introduced. This half precision (16-bit) floating-point data type provides 2x more compact data representation than the single precision floating-point (32-bit) format, but sacrifices data range and accuracy. This type of data is widely used in graphics and imaging applications and reduces dataset size and memory bandwidth consumption.

Next modification made on the core level is new divider. Ivy Bridge introduces a partly pipelined implementation of the divide unit, which is reducing the latency and increasing the throughput of the divide and square root instructions.

Some forms of register-to-register MOV instructions are executed in Ivy Bridge microarchitecture do not consume resources of execution unit. Using these MOV instructions remove instruction latency and frees up the execution unit for other operations running on the machine, improving application performance.

Table 1: Processors characteristic.

Processor type	3rd gen. Intel Core i7-3770K	2nd gen. Intel Core i7-2600	Intel Core i7-970	Intel Core i7-870
Technology (nm)	22	32	32	45
Code name	Ivy Bridge	Sandy Bridge	Gulftown	Lynnfield
Cores per socket	4	4	6	4
Turbo Mode	Yes	Yes	Yes	Yes
Intel HT	Yes	Yes	Yes	Yes
Core Freq. (MHz)	3500	3400	3200	2930
Max Turbo Freq. (MHz)	3900	3800	3460	3600
L1 cache size	32 KB Inst	32 KB Data	per core	
L2 cache size		256 KB	per core	
L3 cache size	8MB	8MB	12 MB	8 MB
Integrated memory controller	Yes	Yes	Yes	Yes
Memory channels	2	2	3	2
Memory type supported	DDR3 1333/	DDR3 1066/	DDR3 800/	DDR3 1066/
(MHz)	1600	1333	1066	1333
Core performance (GFLOPS)	28	27.2	12.8	11.7
Memory Bandwidth (GB/s)	25.6	21	25.6	21
CPU Peak Performance (GFLOPS)	112	108.8	76.8	46.8
Core Turbo performance (GFLOPS)	31.2	30.4	13.8	14.4
Peak Turbo CPU performance (GFLOPS)	121.6	112	80	51.2
TDP (W)	77	95	130	95

For reduction of power Ivy Bridge development team implements DRAM and I/O power gating mechanism. This allows turning off the DRAM interface when it is not in use.

Ivy Bridge has a number of other system enhancements such as support for up to three monitors, dynamic (no reboot required) overclocking control and DDR3LV memory support. The DDR3LV DRAM uses a 1.35V memory module voltage of versus 1.5V for regular DDR3 DRAM this cuts power by 20%.

In this study four generations of the CPUs have been evaluated: the 3rd generation Intel Core i7-3770K processor versus its predecessors the 2nd generation Intel Core i7-2600, six core Intel Core i7-970 and quad Intel Core i7-870. The architecture summary of all processors has been summarized in Table 1.

All CPUs are based on the same microarchitecture foundation but the 3rd and 2nd generations Intel Core processor family have brought enhancements not only in to the instruction set like AVX but also modification in microarchitecture of the core.

3. System Performance

The main focus of this section is to present a comparison of 4 platforms based on CPUs described above. The two first platforms are utilizing AVX enabled CPUs. The last two platforms are utilizing Intel Core CPUs first Intel Core i7-970 processor and second one Intel Core i7-870 processor. These platforms are only limited to the SSE 4.2 instruction set but exploit more cores and different micro architecture implementation.

In this study for evaluation of CPU performance we use a matrix to matrix multiplication micro-benchmark, HPC Challenge Benchmark (HPCC) and NAS Parallel Benchmarks (NPB). Theoretical performance of all tested platform has been summarized in Table 1.

Single core theoretical performance for all platforms is in the range from: 11.7 GFLOPS to 28 GFLOPS. The 139% performance improvement accomplished by Intel 3rd generation Intel Core i7-3770K processor with AVX enabled is mainly achieved due to AVX. For Intel Turbo Boost Technology single core performance we have theoretical performance of 31.2 GFLOPS for Intel 3rd generation Intel Core i7-3770K and 13.8 GFLOPS for Intel Core i7-970, 14.4 GFLOPS for Intel Core i7-870 and 30.4 GFLOPS for Intel 2nd generation Intel Core i7-2600. The difference is 126% between Intel Core i7-970 and Intel 3rd generation Intel Core i7-3770K. The difference in obtained results with Intel Turbo Boost Technology is determined by the different scheme and characteristic of turbo mode implementation. The 3rd and 2nd generations of Intel Core CPUs because of turbo mode allows increased frequency about 4 steps (4 x 100 MHz) for a single core and for Intel Core i7-970 it is 2 steps but (2 x 133MHz) and Intel Core i7-870 can do 5 steps 133MHz each so it increases clock up to 3600MHz. Calculated theoretical performance of all cores, will give 108.8 GFLOPS and 112 GFLOPS for Intel 2nd generation Intel Core i7-2600 and Intel 3rd generation Intel Core i7-3770K respectively. For Intel Core i7-870 we have 46.8 GFLOPS and 76.8 GFLOPS for Intel Core i7-970. Consequently theoretical performance with Intel Turbo Boost Technology will give 121.6 GFLOPS for Intel 3rd generation Intel Core i7-3770K, 112 GFLOPS for Intel 2nd generation Intel Core i7-2600 51.2 GFLOPS and 80 GFLOPS for Intel Core i7-870 and Intel Core i7-970.

Theoretical performance does not completely reproduce the real life scenario. To evaluate how all the new technology enhancements are appropriate for HPC workload we have selected a couple of benchmarks. For the single core performance evaluation we started with a micro-benchmark matrix to matrix multiplication [1]. The size of the matrix used in our study is 2048 x 2048. Table 2 summarizes achieved results for matrix to matrix multiplication micro-benchmark.

Table 2: Matrix to matrix multiplication micro-benchmark.

Processor type	3rd gen. Intel Core i7-3770K	2nd gen. Intel Core i7-2600	Intel Core i7-970	Intel Core i7-870
Single core (GFLOPS)	18	16.3	9.0	9.8
Single core MKL (GFLOPS)	30.1	27.2	12.4	13.3
All cores OpenMP (GFLOPS)	63.1	57.9	36.7	31.7
All cores MKL (GFLOPS)	98.7	89.4	66.5	46.3

For the single core 3rd generation Intel Core i7-3770K processors with turbo mode enabled results are 18 GFLOPS and 16.3 GFLOPS for Intel Core i7-2600 and 9 GFLOPS and 9.8 GFLOPS for Intel Core i7-970 and Intel Core i7-870 respectively. The 3rd generation Intel Core i7-3770K single core performs is 10% better then single core Intel Core i7-2600 and 83% better versus Intel Core i7-870 and 100% better then Intel Core i7-970. When we compiled the same code with MKL the single core 2nd generation Intel Core i7-2600 achieves 11% weaker performance versus the 3rd generation Intel Core i7-3770K and 142%, 126% lower results for Intel Core i7-970, Intel Core i7-870. When we test all cores with OpenMP API we achieved results as follow 63.1 GFLOPS and 57.9 GFLOPS for the 3rd generation Intel Core i7-3770K and 2nd generation Intel Core i7-2600 and 36.7 GFLOPS and 31.7 GFLOPS for Intel Core i7-970 and Intel Core i7-870. Testing all cores with MKL does 98.7 GFLOPS and 89.4 GFLOPS for the 3rd generation Intel Core i7-3770K and 2nd generation Intel Core i7-2600 respectively and 66.5 GFLOPS and 46.3 GFLOPS for Intel Core i7-970, Intel Core i7-870. Achieved results represent 88%, 89%, 83% and 90% of theoretical peak performance, respectively for the 3rd generation Intel Core i7-3770K, 2nd generation Intel Core i7-2600, Intel Core i7-970 and Intel Core i7-870. This matrix to matrix multiplication micro-benchmark shows very significant results even without any hands-on optimizations.

The next benchmark suite we have tested is HPC Challenge benchmark. This is a set of tests that examines the performance of HPC architectures in a more challenging way than small matrix to matrix multiplication micro-benchmark. The HPC Challenge benchmark test suite stresses not only the processors, but also the memory system and interconnects. It is a better indicator of how HPC system will perform across a spectrum of real-world applications, unfortunately some of the test are ineffective in our testing scenario as HPC Challenge benchmark is optimized for complete installation

of supercomputer and clusters and not perfectly scaling down to single socket platform. However a few of the tests which are a good indicator for full system implementation are also suitable to estimate the single platform performance. The HPC Challenge benchmark consists of basically 7 tests: HPL, DGEMM, STREAM, PTRANS, FFT, Random Order Ring Bandwidth and Random Ordered Ring Latency [2]. In this section we will evaluate single platform based on the 3rd generation Intel Core i7-3770K, the 2nd generation Intel Core i7-2600 versus Intel Core i7-870 and Intel Core i7-970 across all the benchmark listed above.

The HPL benchmark is the portable version of LINPACK. This is a floating-point benchmark which solves a dense system of linear equations in parallel. The metric produced is Giga-FLOPS (GFLOPS) or billions of floating point operations per second. LINPACK performs operations called LU Factorization. These are highly parallel and store most of their working data set on the processors cache. It makes relatively few references to memory for the amount of computation it performs [3]. For floating-point performance of the platform we have selected LINPACK as it is most popular HPC benchmark. LINPACK is benchmark used to determine the performance of world's fastest computers published at the website http://ww.top500.org/. The fig. 1 shows HPL results for all tested systems.

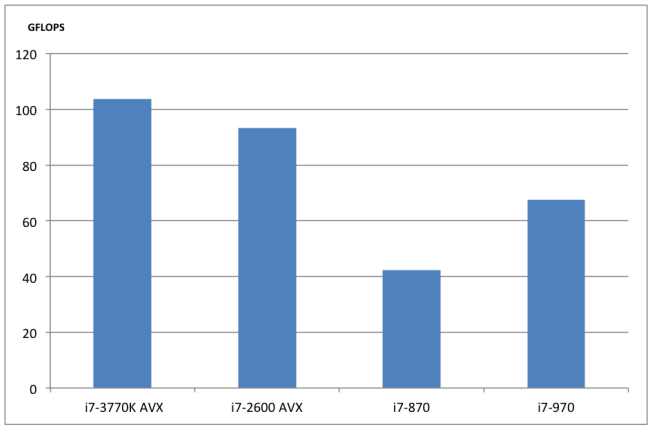


Fig. 1: LINPACK benchmark results.

Single system performance on LINPACK for Intel Core i7-870 is 42.35 GFLOPS, Intel Core i7-970 processor is 67.53 GFLOPS for 2nd generation Intel Core i7-2600 is 93.9 GFLOPS and for the 3rd generation Intel Core i7-3770K this is 103.7 TFLOPS. The achieved performance is 82%, 84%, 83% and 92% of the theoretical performance for Intel Core i7-870, Intel Core i7-970, 2nd generation Intel Core i7-2600 and 3rd generation Intel Core i7-3770K, processor. Intel Core i7-970 has 50% more cores than quad cores then other CPUs processor but AVX version of 3rd and 2nd generation of Intel Core processor have 38% and 54% better LINPACK result. Difference is 120% compeering Intel 2nd generation Intel Core i7-2600 processor to quad cores Intel Core i7-870 and 145% better results versus the 3rd generation Intel Core i7-3770K

The second aspect of platform performance we have evaluated is bandwidth. The benchmark we have used to measure the bandwidth is the STREAM benchmark. It estimates both memory reads and memory writes. This gives us an indication of how effective the memory subsystem is. It measures the performance of four long vector operations. These operations are representative of long vector operations and the array sizes are defined in that way so that each array is larger than the cache of the processors that are going to be tested The STREAM benchmark gives only results of memory system efficiency ignoring the cache efficiencies of the tested platforms. The 3rd generation Intel Core i7-3770K processor based platform has achieved 19.1 GB/s for STREAM using the fastest memory module of DDR3-1600 MHz. The results we have achieved for STREAM are very similar to Intel Core i7-970 18.9 GB/s only 1% difference. The Intel 2nd generation Intel Core i7-2600 processor achieved 17.8 GB/s so 7% weaker than the 3rd generation Intel Core i7-3770K processor. Difference between Intel Core i7-870 and Intel Core i7-970 are more noticeable 16.9 GB/s and 18.9 GB/s respectively. The achieved performance is 75% of theoretical memory bandwidth for 3rd generation Intel Core i7-3770K processor and 84% for Intel 2nd generation Intel Core i7-2600 processor based platforms, 80% Intel Core i7-870 and 73% for Intel Core i7-970. We achieve 73% of theoretical bandwidth result when using 3 channels and 6GB of DDR3-1066MHz on the Intel Core i7-970 based platform, this configuration offers only 1GB memory per core. For other platforms we have dual channel configuration with 8GB memory of DDR3-1333MHz or 1600 MHz and 2GB memory per core ratio.

DGEMM measures the floating point rate of execution of double precision real matrix-matrix multiplication. Fig. 2 shows performance of embarrassingly-parallel DGEMM for the four systems.

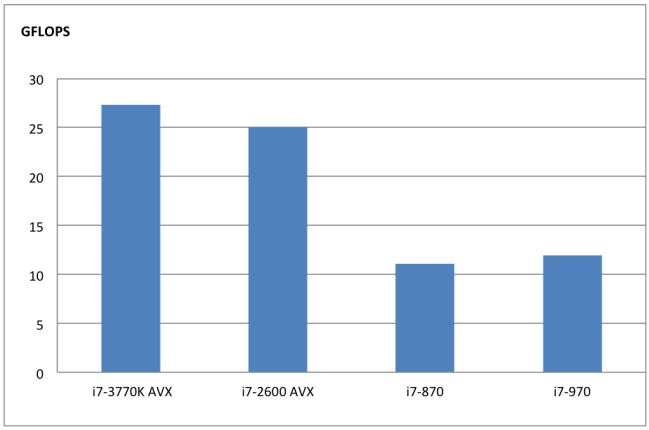


Fig. 2: DGEMM benchmark results.

The 3rd generation Intel Core i7-3770K processor based platform has the highest result 27.35 GFLOPS. The 2nd generation Intel Core i7-2600 processor based platform achieves 25.5 GFLOPS, Intel Core i7-970 processor based platform 11.95 GFLOPS and Intel Core i7-870 processor based platform 11.09 GFLOPS. The achieved performance by 3rd generation Intel Core i7-3770K processor based platform achieved 7%, 128%, and 146% better results than 2nd generation Intel Core i7-2600 processor, Intel Core i7-970 and Intel Core i7-870 processor based platforms respectively.

For FFT benchmark on all four systems we used Intel Compiler 12.0.0.084 as well as the Intel Math Kernel Library 10.3. The benchmark performs as mixture of flops, memory, and network bandwidth. The system based on 3rd generation Intel Core i7-3770K processor and 2nd generation Intel Core i7-2600 processor based platform have the same microarchitecture foundation but slightly different clock and memory subsystem DDR3-1333MHz versus DDR3-1600MHz. Thee difference is 15% but 91% better results versus Intel Core i7-970 and 76% Intel Core i7-870.

The PTRANS benchmark performance depends on the network and on memory bandwidth. The system based on 3rd generation Intel Core i7-3770K has more advance memory implementation and over performs 2nd generations Intel Core i7-2600 processor based system by 11%. Intel Core i7-870 achieved 19% and Intel Core i7-970 45% inferior results versus 3rd generation Intel Core i7-3770K processor based platforms.

The Random Access benchmark uses SANDIA_OPT2 algorithm as Giga Updates per second (GUP/s). All systems have similar memory bandwidth as it is no interconnecting element associated to this platform so results on all platforms are almost the same.

The random-ordered ring bandwidth (RORB) benchmark measures conflict in the network and reports bandwidth achieved per core in a ring communication model. The ROR bandwidth measures the accumulated bandwidth of the communication network of parallel computing systems. The algorithm uses an average, short and long messages transferred with different bandwidth values. Because 3rd generation Intel Core i7-3770K processor based platform has no interconnect element difference in the topology of communication network does not exist consequently the results are very comparable with results achieved by 2nd generations Intel Core i7-2600 processor based system and difference is very marginal. Intel Core i7-870 has achieved 67% lower results versus Intel Core i7-3770K processor based platform and Intel Core i7-970 even 172% below result of Intel Core i7-3770K processor based platform.

The random-ordered ring latency (RORL) measures the latency of the communication network of parallel and, or distributed HPC systems. Several message sizes, communication patterns and methods are used. The algorithm uses an average to take into account that short and long messages are transferred with different bandwidth values in real applications and this has consequences in latency as well [4]. As we have only single systems the relevance of this benchmark is minimal.

The last benchmark suite we have evaluated is the Numerical Aerodynamic Simulation (NAS) Parallel Benchmarks code. This benchmark has been developed by NASA Ames Research Center for the performance evaluation of supercomputers. The benchmark is based on code specifically developed in NASA computational fluid dynamics (CFD) applications department and represents real life applications codes used by NASA for simulating aerospace vehicle.

NAS Parallel Benchmarks consists of two major components five parallel kernel benchmarks and three simulated application CFD benchmarks. The simulated application benchmarks combine several computations defined by the problem size and memory requirements. New revisions of NPB have added three more benchmarks and new versions of problem size for small memory systems. In our study we have focused on the eight tests from the first implementation of NPB as the new added tests, Unstructured Adaptive, Data Cube operator and Data Traffic are not relevant for our testing scenario where we tested only one preproduction platform with single socket. Each benchmark runs two problem sizes Class A and Class B. Same benchmarks also run Class S of the problem size but NASA does not recommend using this problem size for benchmarking purposes [5].

Achieved problem size (Class A) results for 3rd generation Intel Core i7-3770K processor based platform show 8%-13% better results versus 2nd generation Intel Core i7-2600 processor based platform and 43%-113% better than Intel Core i7-870 and 56%-97% better then Intel Core i7-970 based systems. Fig. 3 shows the difference between the 3rd generation Intel Core i7-3770K processor based platform and the platform based on 2nd generation Intel Core i7-2600 processor and additionally compares the Intel Core i7-870 and Intel Core i7-970 based platforms.

Generally platform based on 3rd generation Intel Core i7-3770K processor performs very well and outperforms those based on Intel Core i7-870 and Intel Core i7-970.

For NPB Class B benchmarks we also observe impressive performance benefit of 3rd generation Intel Core i7-3770K processor based platform versus three other configurations. Fig. 4 shows the results from the 3rd generation Intel Core i7-3770K processor based platform versus platform based on 2nd generation Intel Core i7-2600, Intel Core i7-870 and Intel Core i7-970 based platforms for NPB Class B problem size.

For the Class B benchmarks we see that 3rd generation Intel Core i7-3700K processor based platform performs 2%-21% better then 2nd generation Intel Core i7-2600 processor and 43%-142% better than Intel Core i7-870 and 38%-91% superior versus Intel Core i7-970 based platform.

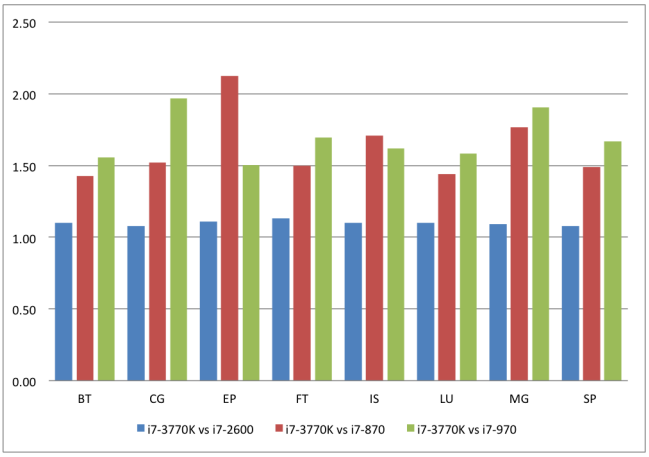


Fig. 3: NPB Class A relative benchmark results.

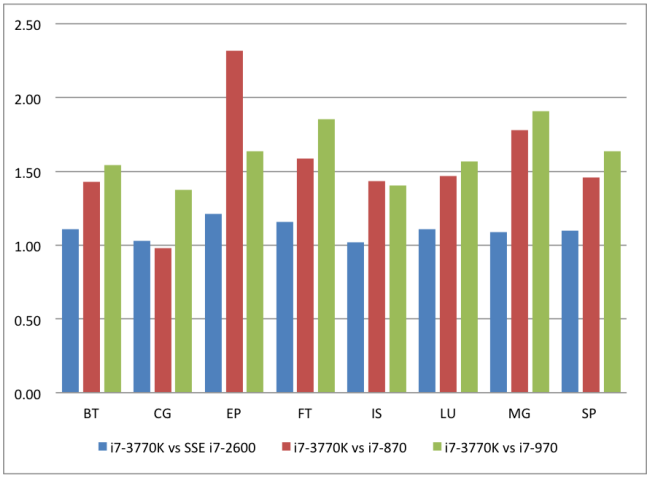


Fig. 4: NPB Class B relative benchmark results.

For IS benchmark 3rd generation Intel Core i7-3770K processor based platform has achieved better result then 2nd generation Intel Core i7-2600 but the difference is marginal only 2%. The Conjugate Gradient (CG) benchmark ran on Intel Core i7-870 got 5% better results than 2nd generation Intel Core i7-2600 processor based platform and even 2% better than 3rd generation Intel Core i7-3770K processor based platform. Overall 3rd generation Intel Core i7-3770K processor based platform confirms its performance superiority.

4. Conclusions

This paper examined the performance characteristics of the Intel 3rd generation Intel Core i7-3770K processor based platform versus its forerunners. In our evaluation study we have selected products which represent the highest CPU configuration in a particular segment and represent the same price point. Of course new generation products typically run on faster clocks and have more capability but our selection was based on the criteria of which product is going to replace its predecessor in a given price segment. So we have compared the same segment platforms from year: 2009, 2010, 2011 and 2012.

The achieved results clearly show that new 3rd generation 22 nm based CPU performs extremely well versus processors based on the Intel Core microarchitecture and results clearly illustrate that AVX instruction set provides major boost in the performance characteristic. The performance improvement we have been able to observe behaves as we have been expecting taking in to account theoretical performance of all CPUs. Intel AVX delivers significant performance improvements to compute-intensive codes and for the same test we have observed 146% difference between then 3rd generation Intel Core i7-3770K processor and its quad core predecessor. For well vectorized code like DGEMM or HPL where AVX can bring benefits the difference is more than 100%, if the code is more memory oriented the difference is smaller 5%-60%. Also six cores Intel Core i7-970 is not able to vie with new 3rd generation Intel Core i7-3770K processor and results show 18%-120% better results.

Performance advantages of 3rd generation Intel Core i7-3770K processor are not only limited to AVX instruction set extension but also the microarchitecture changes and new platform extensions like DDR3-1600MHz brought real performance improvement.

Intel Turbo Boost Technology was always active and delivers additional frequency improvement and consequently improvement in performance for all platforms.

In summary, we can state that 3rd generation Intel Core i7-3770K processor even utilizing a system configuration not optimized for HPC brings lot of performance improvement for compute intensive applications and will deliver a compelling platform for many of the new HPC installations.

5. Acknowledgements

We gratefully acknowledge the help and support provided by Jamie Wilcox from Intel EMEA Technical Marketing HPC Lab.

References

[1] Hubbard, R., Mehrotra, P.: Benefits of Intel AVX For Small Matrices http://software.intel.com/en-us/articles/benefits-of-intel-avx-for-small-matrices/

[2] HPC Challenge Benchmarks, http://icl.cs.utk.edu/hpcc/

[3] Dongarra, J., Luszczek, P., Petitet,A.: Linpack Benchmark: Past, Present, and Future, http://www.cs.utk.edu/ luszczek/articles/hplpaper.pdf.

[4] Barker, K. J. Davis, K. Hoisie, A. Kerbyson, D. J. Lang, M. Pakin, S. Sancho, J.C.A.: Performance Evaluation of the Nehalem Quad-core Processor for Scientific Computing, *Parallel Processing Letters, Vol. 18, No. 4 2008*

[5] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter R., Dagum, L.: The NAS Parallel Benchmarks, http://www.nas.nasa.gov/News/Techreports/1994/PDF/RNR-94-007.pdf

HPC Usage Behavior Analysis and Performance Estimation with Machine Learning Techniques

Hao Zhang[1], **Haihang You**[2], **Bilel Hadri**[2], **and Mark Fahey**[2]
[1]Department of Electrical Engineering and Computer Science,
University of Tennessee, Knoxville, TN 37996, USA
[2]National Institute for Computational Sciences,
Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

Abstract—*Most researchers with little high performance computing (HPC) experience have difficulties productively using the supercomputing resources. To address this issue, we investigated usage behaviors of the world's fastest academic Kraken supercomputer, and built a knowledge-based recommendation system to improve user productivity. Six clustering techniques, along with three cluster validation measures, were implemented to investigate the underlying patterns of usage behaviors. Besides manually defining a category for very large job submissions, six behavior categories were identified, which cleanly separated the data intensive jobs and computational intensive jobs. Then, job statistics of each behavior category were used to develop a knowledge-based recommendation system that can provide users with instructions about choosing appropriate software packages, setting job parameter values, and estimating job queuing time and runtime. Experiments were conducted to evaluate the performance of the proposed recommendation system, which included 127 job submissions by users from different research fields. Great feedback indicated the usefulness of the provided information. The average runtime estimation accuracy of 64.2%, with 28.9% job termination rate, was achieved in the experiments, which almost doubled the average accuracy in the Kraken dataset.*

Keywords: Performance Prediction, Usage Pattern Analysis, Recommendation System, User Support, Machine Learning

1. Introduction

High Performance Computing (HPC) is becoming increasingly important by addressing various applications related to science, engineering, service, commerce, security and defense [15]. To improve productivity, more laboratory researchers are starting to use supercomputers to solve large problems in their research fields. However, most researchers without a computer science background or help from HPC consultants have difficulties productively using the supercomputing resources. Programming for a large HPC system requires more experience than for a desktop computer. First, a user needs to know the system-specific information, such as what software packages are supported by the HPC system and how many compute cores are appropriate for a specific job on the system. Furthermore, to submit a job to an HPC system, a user is usually requested to estimate the runtime of a job for system scheduling. In general, the job runtime estimate is very inaccurate. According to the historical usage data of the world's fastest academic Kraken supercomputer, the average accuracy of the estimated runtime is less than 38% for each job category, and around 32.5% for the entire dataset, which is shown in Figure 1. An inaccurate estimate always causes negative effects. An underestimation increases the risk that a job is forcefully terminated by an HPC system before its completion. On the other hand, an overestimation of the job runtime usually results in a longer queuing time. In both cases, the productivity of an HPC user is damaged.

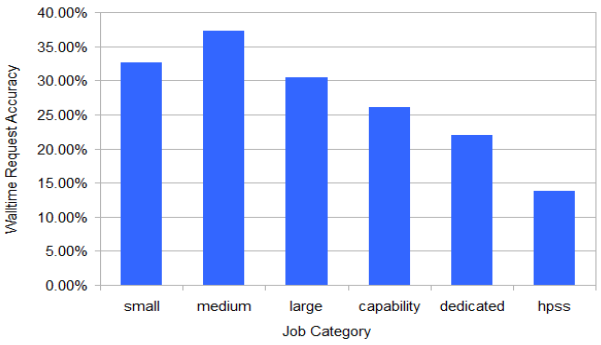


Fig. 1: Average accuracy of the requested walltime for each job category in Table 1.

In this work, we investigated the usage behavior patterns of the Kraken supercomputer, which is the world's fastest petascale academic supercomputer and the 11th on the latest Top500 list. We also developed a knowledge-based recommendation system to optimize user service and resource allocation with the purpose of improving user productivity at the application level. Historical usage logs of the Kraken supercomputer were used to analyze the underlying usage patterns, which were collected from January 2, 2011 to July 21, 2011. We implemented six unsupervised machine learning algorithms to identify usage behaviors, and applied three validation measures to compare the clustering algorithms and determine the appropriate number of behavior categories.

Table 1: Manual classification of jobs in the Kraken supercomputer, which is determined by the number of compute nodes requested by a user.

Categories	Compute Cores			$\text{Walltime}_{\text{max}}$
	Min	Max	Percentage	(hours)
Small	1	512	0-0.45	24.0
Medium	513	58,192	0.45-7.26	24.0
Large	8,193	49,536	7.26-43.88	24.0
Capability	49,537	98,352	43.88-87.12	48.0
Dedicated	98,353	112,896	87.12-100	48.0
HPSS	N/A	N/A	N/A	24.0

The correct jobs were grouped into seven categories, each of which represented a usage behavior of an HPC system. The clustering results can be used by an HPC center to better understand its user community and provide customized services to different groups of users with different types of job submissions. Moreover, considering the fact that HPC systems with different architectures perform differently on the same type of jobs, performance comparisons across different HPC systems provide the potential to identify which architecture is superior to execute which type of jobs. Then, based on the statistics in each behavior category, a knowledge-based recommendation system was developed to provide users with instructions about choosing software packages, setting job parameter values, and predicting job performance. The proposed system enables research communities to share usage experience of a supercomputer, and researchers may improve their productivity by looking at similar jobs successfully executed by other users in the same research area on the same HPC system.

The rest of the paper is organized as follows. Section 2 reviews existing work. Section 3 describes the historical usage data of the Kraken supercomputer, and discusses data preprocessing and feature extraction methods. The unsupervised machine learning algorithms and cluster validation measures are introduced in Section 4 for usage behavior analysis. The proposed recommendation system is also described in this section. Then, Section 5 presents the experimental results on Kraken. Finally, Section 6 concludes the paper.

2. Related Work

Although workload modeling was widely researched [19], only a few studies were reported to address the task of usage behavior analysis. A naive approach to solve this problem is to manually classify a job, based on the number of the requested compute nodes of the job, which is adopted by most HPC centers for job scheduling, such as the Kraken supercomputer [11] as shown in Table 1. Another statistical approach was introduced in [7] to characterize job behaviors and identify the most active users. Wolter [18] experientially classified supercomputer users into three groups. Song [14] proposed a mixed user group model to classify HPC users and analyze workload traces for job scheduling. However, the number of the usage categories was predefined, and the author did not explicitly validate the classification results.

Another related work is to predict the runtime of a job using historical data. The most widely used methods, such as in [10], [16], were directly based on the instances in the historical usage data, which estimated job runtime using the average runtime of a set of jobs with the highest similarity. However, a petascale supercomputer usually has millions of job submissions. It becomes inefficient or even infeasible for instance-based approaches to save all instances. Some model-based algorithms were also proposed for runtime prediction, including methods using artificial neural networks [8] and template-based approaches with greedy and genetic algorithms [13]. However, artificial neural network, as a black box model, failed to explicitly unveil the underlying structures of usage behaviors. In template-based approaches, the templates had to be defined manually, which are almost infeasible for a dataset with millions of samples.

3. Usage Data Processing

In this work, we looked at the historical usage data of the Kraken supercomputer [11], which is managed by the National Institute for Computational Sciences (NICS) at the Oak Ridge National Laboratory. Kraken provides a petascale computing environment fully integrated with the Extreme Science and Engineering Discovery Environment with access to the Cray XT5 system. The Kraken supercomputer consists of 18,816 compute sockets, 147 terabytes of memory, and 3.3 petabytes of storage. Access to computing resources is managed by the Portable Batch System, and job scheduling is handled by the Moab. The Lustre file system is used to support I/O operations.

Usage behavior patterns were analyzed mainly based on the historical data of jobs that were submitted to the Kraken supercomputer. Some instances of usage log entries are listed in Table 2. We also collected account information to determine to which research field each user belongs, and conducted surveys to gather specific information, such as whether a user was an HPC expert. Because most users are not HPC experts or with help from HPC consultants, it is unavoidable that their programs contain runtime errors and exit before completion. On the other hand, due to the lack of experience to run a job on a supercomputer, it is also very common that users underestimate job runtime, and a job is forcefully terminated by the HPC system. If a job neither completes correctly nor provides meaningful results, it is considered as an incorrect job. On the contrary, a job is defined *correct*, if it belongs to the set:

$$\mathcal{J}_c = \{j \mid (j.mem\_used \neq 0) \wedge (j.walltime > 30) \wedge (j.walltime < j.walltime\_req)\} \quad (1)$$

where "." represents attribute relationship, and the unit of the attribute *walltime* is second. Intuitively, a job is incorrect if

Table 2: A typical log file that records the usage activities of the Kraken supercomputer. This exemplary log file contains five instances. Each row represents a job submission, and each column denotes an attribute of a job.

job_id	user_name	account	nproc	mem_used	submit_time	start_time
0000001.nid00016	user1	U1-INDEX001	12	7588	2011-01-27 08:10:13	2011-01-27 09:26:03
0000002.nid00016	user2	U2-INDEX001	1	142664	2011-03-28 14:15:50	2011-03-28 14:16:14
0000003.nid00016	user1	U2-INDEX001	1032	16276	2011-04-16 01:28:41	2011-04-16 11:51:30
0000004.nid00016	admin	SUPPORT	288	11836	2011-05-31 09:43:51	2011-06-04 21:00:20
0000005.nid00016	user1	U1-INDEX002	98304	71812	2011-07-05 17:01:08	2011-07-07 11:11:13

end_time	walltime_req	walltime	cpu_hours	queue	type	software	research_area
2011-01-27 09:36:14	00:30:00	00:10:11	3.22	small	batch	——	Physics
2011-03-28 14:35:01	05:30:00	00:18:47	0.0658	hpss	batch	wrf	Earth Sciences
2011-04-17 11:51:56	24:00:00	24:00:27	24775.74	medium	batch	mpcugles	Physics
2011-06-05 21:00:57	23:59:59	24:00:37	6914.96	small	interactive	——	Benchmark
2011-07-08 13:11:34	32:00:00	26:00:21	2556477.44	capability	batch	gadget	Earth Sciences

1) it doesn't consume any memory, which indicates that the job fails to start executing, possibly due to some compiling problems, including the situation that the job is compiled on another HPC system with a different architecture; or 2) the job quits right after its execution, which is possibly caused by runtime errors, such as segmentation faults; or 3) the job is terminated by the HPC system, because it consumes the requested walltime. The second literal in (1) also contributes to remove the "Hello World" jobs and simple testing jobs, in which case a user is a learner or does not care about job runtime. Because incorrect jobs are always misleading, it is necessary to remove them to improve the performance of job classification and runtime estimation. On the other hand, it should be noted that the definition of a correct job does not remove all incorrect jobs, such as a job with runtime errors at the end of its execution. The rest of the incorrect jobs are considered as noise, or instances with errors.

A feature is a distinctive characteristic of an object, such as a job, which is often represented as a function of the object's attributes. In this work, four features are selected or extracted from job attribute space, which are defined as:

$$\begin{aligned} N_n(j) &= \log(\, j.nproc \,/\, C_n \,) \\ M_u(j) &= \log(\, j.mem\_used \,) \\ T_q(j) &= \log(\, j.start\_time - j.submit\_time \,) \\ T_r(j) &= \log(\, j.end\_time - j.start\_time \,) \end{aligned} \tag{2}$$

where N_n, M_u, T_q and T_r are the features encoding the information of number of allocated nodes, memory used, job queue time and job runtime, respectively; and C_n represents the number of cores on each node, which is 12 on the Kraken supercomputer. Because job attributes cover a large range of values, logarithmic scale is used to represent each feature to reduce a wide range to a manageable and comparable size.

4. Usage Pattern Modeling

We remove the jobs submitted by system administrators that typically perform system management tasks, such as system updating or reservation removing. Moreover, jobs in the HPSS queue and interactive jobs are also intentionally removed. HPSS jobs often start executing right after being submitted, without any compute nodes involved. Interactive jobs provide a user interactive access to compute resources, which are commonly used for debugging. At last, we manually classify the jobs requiring over 50,000 cores into the group of massive jobs. According to our survey, all users with very large job submissions are HPC experts or work with HPC consultants. Formally, the job dataset used in this work is defined as:

$$\begin{aligned} \mathcal{J} = \{\, j \mid (j \in \mathcal{J}_c) \wedge (\, j.nproc < 50000 \,) \\ \wedge (\, j.queue \neq \text{hpss} \,) \wedge (\, j.type \neq \text{interactive} \,) \,\} \end{aligned} \tag{3}$$

4.1 Cluster Analysis and Validation

To investigate the underlying structures of usage patterns, cluster analysis is applied on job set $\mathcal{J}$. Cluster analysis is an unsupervised learning technique, which groups objects with similar attributes into respective categories. In cluster analysis, we do not know either which job belongs to which category or the number of categories. Six clustering methods, in three categories, are applied to analyze usage behaviors:

- Partitioning clustering, including k-means [1] and Partitioning Around Medoids (PAM) [17];
- Hierarchical clustering, including DIvisive ANAlysis clustering (DIANA) [5] and Unweighted Pair Group Method with Arithmetic Mean (UPGMA) [9];
- Artificial Neural Network (ANN) based clustering, including Self-Organizing Feature Map (SOFM) [6] and Self-Organizing Tree Algorithm (SOTA) [4].

Cluster validation evaluates the performance of a clustering result by comparing it with other ones that are generated

by other clustering methods, or by the same method but with different parameters, such as different number of clusters. In this work, three internal validation measures, along with domain knowledge, are used to choose the best clustering methods and determine the number of clusters that is most appropriate for the dataset:

- Connectivity [2] measures cluster connectedness, with a value in the range $[0, \infty]$. A lower value indicates a better performance.
- Dunn index [3] is the ratio of the smallest distance between objects in different clusters to the largest intra-cluster distance. The value of Dunn index lies in $[0, \infty]$, and a greater value indicates a better performance.
- Silhouette width [12] measures the degree of confidence in the clustering assignment of an object. Its value lies in $[-1, 1]$ with a greater value indicating a better clustering performance.

4.2 Information Recommendation

It is clear that jobs have a wide range of system demands and might perform differently on the same HPC system. For example, computation-intensive and data-intensive jobs might have significantly different runtime, even if they request the same number of compute nodes. The proposed knowledge-based recommendation system is based on the plausible assumption that job submissions in the same research field tend to perform more similarly, since jobs submitted by researchers in the same field are more possible to address similar problems with similar algorithms using similar software and packages. Moreover, jobs in the same clustered categories also tend to behave more similarly due to the similarity of their attributes. In consideration of both assumptions, we predict the parameters of a target job based on the information of the jobs that are in the same category and same research field as the target job.

In the first step, customized information is provided to a user by the recommendation system, including the distribution of the jobs, the statistical information of several job features, and a list of the most frequently used software and packages in the given research field. Jobs in a research field conforms to the multinomial distribution given job categories. A bar graph is used to intuitively represent this distribution. Then, the statistical summaries of several features are graphical displayed using box plots. Each box plot represents the distribution of a feature, its central value, and variability. The displayed features include N_n and M_u, i.e., the number of requested compute nodes and consumed memory. At last, a list of software and packages is provided, which is obtained by ranking the most frequently used software and packages used by previous users in the same research field. In this step, a user is required to determine the number of requested cores and memory, under the guidance of the recommendation system.

In the second step, after the number of requested cores and memory are decided, the features N_n and M_u are computed according to (2). Then, the queuing time and runtime of the target job are predicted based on the job distributions and the job statistics in each behavior category. More precisely, our goal is to predict the feature vector $\boldsymbol{z}_j = \{T_q, T_r\}$. Let $\boldsymbol{y}_j = \{M_u, Nn\}$ be the feature vector that is determined in the first step by a user in field r_j. Then, the distance between $\boldsymbol{y}_j$ and the center of the behavior category k can be computed using Euclidean distance: $d_{jk} = \|\boldsymbol{y}_j - \mathrm{E}[\boldsymbol{y}_{jk}]\|$, where $\mathrm{E}[\boldsymbol{y}_{jk}]$ is the feature vector expectation of the jobs in the behavior category k and the research field r_j. Then, the target feature vector $\boldsymbol{z}_j$ can be computed by:

$$\boldsymbol{z}_j = \frac{1}{Z_j(\boldsymbol{y}_j, \boldsymbol{\theta}_j)} \sum_{k=1}^{K} \frac{\theta_{jk}}{d_{jk} + \sigma} \mathrm{E}[\boldsymbol{z}_{jk}] \tag{4}$$

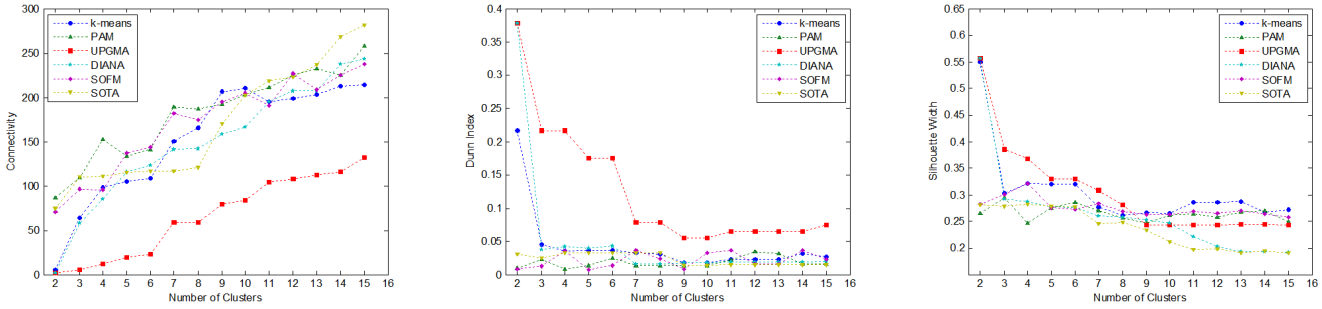
where θ_{jk} is the probability that job j belongs to category k; σ is a small positive number to avoid a zero denominator; and $Z_j(\boldsymbol{y}_j, \boldsymbol{\theta}_j)$ is a normalizing factor. The feature vector $\boldsymbol{z}_j$ is computed using a mixture of corresponding cluster means. The mixture coefficient considers both the clustering result of the target job, which is represented by the distance to each cluster center, and the prior knowledge $\boldsymbol{\theta}_j$ that is the distribution of the jobs in the target job's research field. Both elements are important for job classification. While the distance of a job to a cluster center determines the probability that the job belongs to the cluster, the job distribution in a research field, as a prior, prefers the cluster containing a larger number of jobs that are previously observed. After the feature vector $\boldsymbol{z}_j$ is calculated, the queuing time can be computed by: $\hat{j.queuing} = 10^{T_q}$. In most cases, users care more about job runtime, which is a required parameter by most supercomputers to submit a job. According to our experiments, the job runtime can be better estimated by:

$$\hat{j.runtime} = 10^{(1+\alpha)T_r + \beta} \tag{5}$$

where $\alpha, \beta \in [0, 1]$. A good job runtime estimation can be computed with $\{\alpha, \beta\} = \{0.05, 0\}$, and a safer estimation can be obtained with $\{\alpha, \beta\} = \{0.05, 0.3\}$, which can be applied to request system walltime. The coefficient $(1 + \alpha)$ of T_r considers the fact that, for a larger job with more compute resources and longer runtime, users often tries to make the job safer (i.e., with a higher probability to successfully complete the job), by intentionally requesting a longer system walltime. If a large job is forcefully terminated by the system before its completion, it often costs much more than a small job.

5. Experimental Results

We used the historical usage data that were collected from January 2 to July 21, 2011, after Kraken was upgraded to the Cray XT5 system. During the time of data collection, 321,290 jobs in 24 different research fields were submitted to



(a) Connectivity measure (b) Dunn index measure (c) Silhouette width measure

Fig. 2: Internal cluster validation of the clustering methods with different number of clusters.

Kraken by 843 researchers from 367 accounts. The Kraken dataset was first preprocessed to get job set $\mathcal{J}$. In this step, 112265 jobs were removed from the original Kraken dataset, including 102206 incorrect jobs (termination-rate = 31.8%). Then, feature vectors were computed for the rest of the jobs according to (2), and related information, including research field and software name, was documented.

5.1 Cluster Analysis and Validation

Before clustering the Kraken dataset, a cluster was manually defined as the massive job category that contains jobs requiring more than 50,000 compute cores, which contains 168 job submissions in the entire preprocessed dataset.

Six clustering approaches were applied to cluster usage behaviors, and three internal validation measures were employed to compare the clustering performance and to find the appropriate number of clusters. The performance evaluation results are depicted in Figure 2a, 2b and 2c, respectively. A most important conclusion is that the UPGMA algorithm performs consistently better than other clustering methods when using less than nine clusters, which is indicated by all internal validation measures. We can also observe that other clustering techniques, if the number of clusters is greater than three, have similar performance but are generally worse than the UPGMA algorithm. Thus, the UPGMA algorithm, an agglomerative hierarchical clustering algorithm, was selected to cluster usage behaviors.

The validation measures were also applied to determine the appropriate number of clusters. For all clustering algorithms, the connectivity measure in Figure 2a indicates that, with the increase of the number of clusters, the clustering performance tends to decrease. Similarly, both the Dunn index in Figure 2b and the silhouette width in Figure 2c indicate that clustering performance cannot be improved by simply increasing the number of clusters. We can also observe that, for the k-means, PAM and UPGMA algorithms, two is the best choice for the number of clusters, which is supported by the connectivity measure and the Dunn index measure, and partially by the silhouette width measure. On the other hand, in order to increase the resolution of the job clustering results, we prefer a larger number of clusters. For the chosen UPGMA algorithm, the clustering performance almost consistently declines with the increase of the number of clusters. But the performance of the algorithm does not change greatly with the number of clusters lying between three and six. To balance clustering performance and clustering resolution, six clusters were selected, in which case, we increased the clustering resolution by sacrificing some clustering performance. The resulted dendrogram with six clusters is depicted in Figure 4.

Table 3: Algorithmic mean of each job feature in different categories in the Kraken dataset.

Feature	c1	c2	c3	c4	c5	c6	c7
E(T_q)	4.24	2.62	2.87	1.89	4.65	3.74	4.27
E(T_r)	4.19	3.63	3.22	2.38	4.15	2.64	3.22
E(M_u)	4.05	4.03	5.11	4.09	4.17	4.12	5.07
E(N_n)	0.19	0.26	1.01	0.98	1.51	1.51	3.90

The algorithmic means of job features in each category are listed in Table 3. Several interesting phenomena should be noted. First, the behavior category c3 can be considered as the set of data-intensive jobs. Jobs in this category use significant memory but relatively less number of compute nodes. Second, the jobs in the manually defined category c7 are greatly different from the jobs in other categories, which are both computational and memory-intensive. However, jobs in this category do not have the longest queuing time, because the massive jobs have a high priority on the Kraken supercomputer, and researchers running such jobs usually reserve the compute resources in advance. These massive jobs do not have the longest job runtime either. Besides the fact that the massive jobs use considerable amount of resources that can speed up job execution, this phenomenon can be partially explained by the fact that most users running such jobs have rich HPC knowledge. In most cases, the massive jobs are optimized. Third, if we only consider the

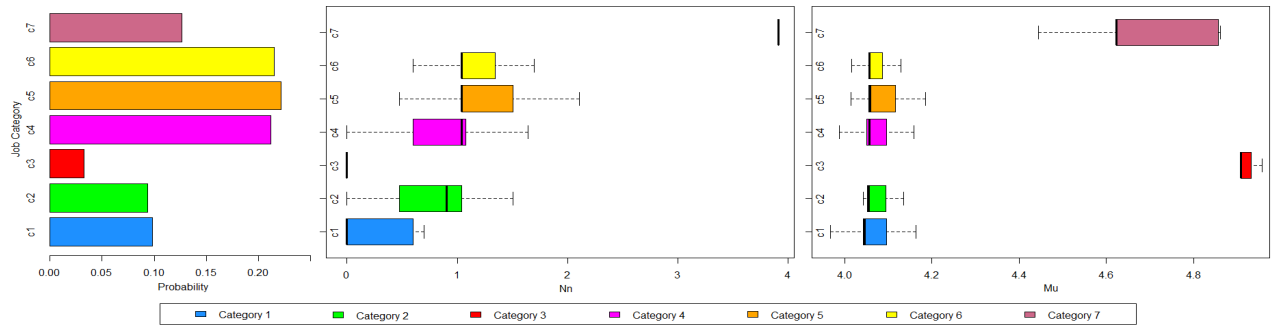


Fig. 3: Customized information of the jobs in the research field of Astronomical Sciences, which is generated in the first-round recommendation. The bar graph (left) represents the distribution of the previously submitted jobs on Kraken. The box plot (center) summaries the statistics of the number of compute nodes requested by the jobs in each category in log scale. The box plot (right) represents the statistical summary of the consumed memory of the jobs in each category in log scale.

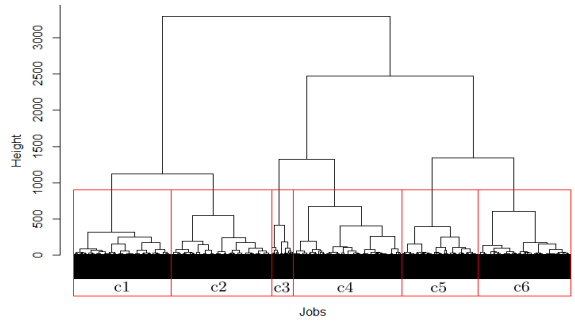


Fig. 4: Dendrogram generated by the UPGMA algorithm with six categories on the Kraken dataset.

requested compute nodes and the consumed memory to classify jobs, we can combine the category pair: c1 and c2, c3 and c4, c5 and c6 in to larger clusters, due to the similarity of the features in each category pair. This phenomenon is well supported by the dendrogram. If the UPGMA algorithm is set to stop with three clusters, the dendrogram will be cut at the height of 2000, as shown in Figure 4. Fourth, jobs in the categories c1 and c2 have the longest runtime. Besides the fact that these jobs use less compute resources, another possible explanation is that users executing such jobs usually have less HPC experience and their jobs are not well-tuned.

5.2 Information Recommendation

To demonstrate how the proposed recommendation system instructs a user to select appropriate job parameters and predict job performance, a real case is used as an example, in which case, a researcher planned to solve a problem in the field of Astronomical Sciences. In the first-round recommendation, the system first provided a list of packages that were most frequently used on Kraken to solve problems in Astronomical Sciences, such as GADGET, YT, SSES, and CHIMERA. After comparing the packages, the researcher decided to use Gadget, which is a cosmological smoothed particle hydrodynamics (SPH) simulator. The proposed system also provided the researcher with customized information about the job statistics in Astronomical Sciences, as shown in Figure 3. Some important phenomena were observed by the researcher. First, more large jobs in c4, c5, c6 and c7 were executed in this field. Second, almost all massive jobs in c7 used considerable resources with similar number of compute nodes but widespread memory. Third, only a few jobs were data-intensive which generally consumed a very small number of compute nodes but significant amount of memory. Fourth, the non-massive non-data-intensive jobs, in c1, c2, c4, c5 and c6, consumed similar amount of memory, as shown by the right box plot. The medians of the number of requested compute nodes of jobs in c4, c5 and c6 were very close, as indicated by the center box plot in Figure 3. Based on these observations, the researcher decided to request 12 GB memory ($\hat{M}_u \approx 4.1$) and 12 compute nodes ($\hat{N}_n \approx 1.1$) for the job.

In the second-round recommendation, the values of T_q and T_r were computed according to (4), given $\hat{M}_u$ and $\hat{N}_n$. The architecture of the recommendation system and the estimating process were transparent to users. All feedback the researcher received from the system was the estimated queuing time, runtime, and a safe runtime, which can be used to request system walltime on the Kraken supercomputer. In this case, the estimated queuing time was 7079 seconds ($z_{Tq} = 3.85$); the estimated runtime was 5624 seconds ($z_{Tr} = 3.57$); and the estimated safe runtime was 11221 seconds. After successfully compiling the program on Kraken, the researcher used the estimated runtime as the requested system walltime to execute the job. After waiting in the queue of type *small* for 11175 seconds on Kraken, the job started executing, and the actual runtime was 5049 seconds, which was smaller than but quite close to the job estimated runtime provided by the proposed recommendation system.

In order to quantitatively measure the accuracy of the runtime estimation, we applied the Walltime Request Accuracy (WRA) for a correct job j in $\mathcal{J}_c$, which is defined as:

$$WRA(j) = \frac{j.walltime}{j.walltime\_req} \times 100\% \quad (6)$$

In the evaluation, 127 jobs submitted by seventeen researchers from different research fields were used to test the proposed system. The researchers were not HPC experts or worked with HPC consultants, and the jobs were moderately optimized. According to their feedback, all researchers considered that the information provided by the recommendation system was helpful, especially the job statistics in a research field. For job runtime estimation, when the researchers were asked to use the estimated runtime provided by the recommendation system as the requested walltime, 28.9% jobs were terminated in force by the Kraken supercomputer, due to running out of requested walltime. For the correct jobs, an average WRA of 64.2% was achieved. Comparing to the average WRA of 32.5% with the termination rate of 31.8% in the Kraken dataset, the estimation performance was greatly improved. If the users were asked to use the safe runtime estimate to request system walltime, only 12.2% jobs exceeded their requested walltime. But the average WRA decreased to 33.3%. In this case, although the resulted average WRA was similar to the average WRA in the Kraken dataset, the termination rate was greatly decreased. In order to quantitatively measure the error of the estimated queuing time, the mean absolute percentage error (MAPE) was applied, which is defined as:

$$MAPE = \frac{1}{T}\sum_{t=1}^{T}\left|\frac{j.\hat{queuing} - j.queuing}{j.queuing}\right| \quad (7)$$

where T is the number of testing instances, and $j.\hat{queuing}$ and $j.queuing$ are the estimated and actual queuing time, respectively. The resulted MAPEs were 67.2% and 105.8% using the estimated runtime and the safe runtime to request system walltime, respectively. This result indicated that the queuing time is harder to estimate, which is heavily dependent on the job queue status. Fortunately, the researchers, like most HPC users, did not much care about the queuing performance according to their feedback.

6. Conclusion

Usage behaviors of the Kraken supercomputer were systematically investigated and a knowledge-based recommendation system was developed to improve job performance and user productivity at the application level. Besides manually defining a category for massive job submissions, six behavior categories were identified with the UPGMA algorithm, which presented the most promising performance on the Kraken dataset. Then, a knowledge-based recommendation system was developed based on the identified behavior categories and job statistics. The proposed recommendation system is able to: 1) provide customized information to help a user determine the software packages, and the number of compute nodes and amount of memory to request for a job, 2) predict job queuing time and runtime, and estimate a safe job runtime to request system walltime. Great feedback from users demonstrated the usefulness of the recommendation system. The average runtime estimation accuracy of 64.2%, along with the job termination-rate of 28.9%, was achieved, which almost doubled the previous average accuracy in the Kraken dataset.

References

[1] K. Alsabti, S. Ranka, and V. Singh, "An efficient space-partitioning based algorithm for the k-means clustering," in *The Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, 1999, pp. 355–359.

[2] L. Deborah, R. Baskaran, and A. Kannan, "Survey on internal validity measure for cluster validation," *International Journal of Computer Science and Engineering Survey*, vol. 1, no. 2, pp. 85–102, Nov. 2010.

[3] J. C. Dunn, "A fuzzy relative of the ISODATA process and its use in detecting compact Well-Separated clusters," *Journal of Cybernetics*, vol. 3, no. 3, pp. 32–57, 1973.

[4] J. Herrero, A. Valencia, and J. Dopazo, "Phylogenetic reconstruction using a growing neural network that adopts the topology of a phylogenetic tree," pp. 226–233, 1997.

[5] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley-Interscience, Mar. 1990.

[6] T. Kohonen, *Self-organized formation of topologically correct feature maps*, 1988, pp. 509–521.

[7] H. Li, D. Groep, and L. Walters, "Workload characteristics of a multi-cluster supercomputer," *Job Scheduling Strategies for Parallel Processing*, vol. 3277, 2004.

[8] J. Li, X. Ma, K. Singh, M. Schulz, B. de Supinski, and S. McKee, "Machine learning based online performance prediction for runtime parallelization and task scheduling," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[9] C. D. Michener and R. R. Sokal, "A quantitative approach to a problem in classification," *Evolution*, vol. 11, 1957.

[10] T. Minh and L. Wolters, "Using historical data to predict application runtimes on backfilling parallel systems," in *Euromicro International Conf. on Parallel, Distributed and Network-Based Processing*, 2010.

[11] NICS, "Running jobs on Kraken," http://www.nics.tennessee.edu/node/16, accessed: 11/11/2011.

[12] P. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *J. Comput. Appl. Math.*, vol. 20, pp. 53–65, Nov. 1987.

[13] W. Smith, I. T. Foster, and V. E. Taylor, "Predicting application run times using historical information," in *Workshop on Job Scheduling Strategies for Parallel Processing*, 1998, pp. 122–142.

[14] B. Song, C. Ernemann, and R. Yahyapour, "User group-based workload analysis and modelling," in *IEEE International Symposium on Cluster Computing and the Grid*, vol. 2, may 2005, pp. 953–961.

[15] Top500, "Application area share for 06/2011," http://www.top500.org, accessed: 11/11/2011.

[16] D. Tsafrir, Y. Etsion, and D. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates," *IEEE Trans. Parallel and Distributed Systems*, vol. 18, no. 6, pp. 789–803, 2007.

[17] M. van der Laan, K. Pollard, and J. Bryan, "A new partitioning around medoids algorithm," *Journal of Statistical Computation and Simulation*, vol. 73, no. 8, pp. 575–584, 2003.

[18] N. Wolter, M. McCracken, A. Snavely, L. Hochstein, T. Nakamura, and V. Basili, "What's working in HPC: Investigating HPC user behavior and productivity," *CTWatch Quarterly*, vol. 2, no. 4A, 2006.

[19] H. Zhang and H. You, "Comprehensive workload analysis and modeling of a petascale supercomputer," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 2012.

SESSION

COMPUTATIONAL SCIENCE + COMPUTAIONAL FINANCE

Chair(s)

TBA

Parallel Implementation of Moving Averages and Stock Market Prediction

John Jenq
Computer Science Department, Montclair State University, Montclair, New Jersey, USA

Abstract - *In recent years, graphics processing units have made parallel processing affordable with the price of personal desktop computers. This report investigates the computational aspects of calculating simple moving average and exponential moving average operations, two of the most popular financial indicators. In this report, we also investigate the usage of GPU to run artificial neural network as a mean of predicting stock market pricing. Feedforward and Backpropagation artificial neural network was used for this study. Financial data including major stock indices, volumes, pricing, and moving average of stocks were used as input. The future stock prices can be predicted as the output. The speedup factor by adopting GPU and CPU together over traditional CPU alone implementation was not significant. The computation of compute moving averages on GPU was also discussed.*

Keywords: artificial neural network, stock prediction, GPU computing, parallel processing, high performance computing.

1 Introduction

Graphic processing unit (GPU) has transformed a regular PC into a personal supercomputer. For example, in [5], the GeForce GTX 580 can perform single precision operation that reaches more than 1500 GFlops. These computing powers significantly speed up the computational intensive applications with a price of a PC. Many tools have been developed to make the GPU computing much easier than ever before. Personal supercomputing is now a reality to us.

Artificial neural networks are used for pattern recognition, clustering, and optimization. Neural networks can also be used to solve problems which are not easily solved by traditional calculation methods, particularly if there is no strong underlying theory to explain the data. Neural networks have been developed as generalizations of mathematical methods of neural biology, based on the assumption that the information processing occurs at many simple elements called neurons. Signals are passed between neurons over connection links. Each connection link has an associated weight which multiplies the signal transmitted. Each neuron applies an activation function to its net input to determine its output signal.

There are researchers using neural network in financial and economic computations. For example, in [3], Li and Liu used LM BP algorithm to predict the Shanghai stock market. In [9], Wang developed a HLP method that gets stock high low points with different frequencies and amplitudes. The extracted data is then fed into a neural network to forecast the stock direction and price. In [8] Tirados and Jenq used neural networks to predict GDP with ten leading economic indicators as the input. In [4] Lin and Feng combined neural network and pattern matching techniques to analyze and forecast oil stock prices. In [10], Zhou and Zhang used financial indicators such as moving averages, volumes, relative strength index, etc. on neural network to predict future stock prices.

In the past, CPU clusters have been used to achieve high performance computation. GPU computing uses GPU as a co-processor to accelerate CPUs for general purpose scientific and engineering computing. It shifts computation intensive program segments into GPU while keeping the rest of the program segments, which are serial in nature, on the CPU. This kind of hybrid computing improves the performance of many computer applications.

The GPU computation can be used on financial computations as well. Researchers in the financial world find the benefits of using GPU in financial computation. In [6], Peng, et. al., compute option pricing on GPU with backward stochastic differential equation. in [1], Lee, et. al., did financial derivative modeling using GPUs. In [7] Solomon et. al., used trinomial lattice strategy to implement the pricing of European option and American lookback option pricing using GPU. In [2], Lee, et. al., investigated random number generation and the Monte-Carlo simulation to predict future stock prices. They also discussed the out of core case when graphics DRAM is not big enough to hold all the application data.

The rest of the paper is organized as the following. Section 2 discusses methodologies and implementation of an artificial neural network. Parallel implementation of simple moving average and exponential moving average will be discussed as well. Section 3 discusses and analyzes the experimental results. Section 4 gives conclusion remarks.

2 Methodologies and implementations

Three-layer neural network was chosen to implement our prediction system. The inputs are major industrial stock indices and stock indicators. The goal is to forecast future stock prices. Feedforward and backpropagation neural network was used. Backpropagation is a gradient descent neural network method used to minimize the total squared error of the output calculated by the network. The network is developed to achieve a balance between the ability to respond to the input patterns that are used for training and the ability to give good responses to input that is similar, but not identical, to that used in training. Like with multiple regression, backpropagation was used to develop a correlation between the input of stock market data in order to determine the future stock price.

The training of a network by backpropagation involves three stages: the feedforward of the input training pattern, the calculation and backpropagation of the associated error, and the adjustment of the weights. After training, application of the network involves the computations of the feedforward phase only.

To prevent the larger data from dominating the outcome, the raw data will be processed before being fed into the neural network. The raw data has been modified. In the following discussion, *B* represents the raw form of the original data, and C is the normalized version of *B*.

$$C_t = 2\left(\frac{B_t - B_{\min}}{B_{\max} - B_{\min}}\right) - 1 \qquad (1)$$

This transformation map our data to between -1 and 1. The activation function selected is the bipolar sigmoid function, which has a range of (-1, 1), and is defined as

$$f(a) = \frac{2}{1 + exp(-a)} - 1 \qquad (2)$$

$$\frac{df(a)}{da} = \frac{1}{2}[1 + f(a)][1 - f(a)] \qquad (3)$$

2.1 Feedforward Backpropagation Neural Network

The traditional Neural Network algorithm can be simplified as below

```
while (cycle < maxCycle  && averageError > toleranceEerr)
{
    for (int i= 0; i < totalRecords;  i++)
    {
        forwardPropagation();
        backwardPropagation();
        accumulateError();
        updateWeights(learningRate);
    }
    compute averageError;
}
```

To simplify the discussion, let's denote the connection weights between the input layer and hidden layer to be *w*. Also let's denote the connection weights between the hidden layer and output layer as *v*. Note that both *w* and *v* are vectors. To parallelize the code in order to fit into GPU computing, some modifications will be made. Instead of performing backpropagate operation for each pattern to update weights, we will compute the *dw* (the update of weight *w*), *dv*(update of weights *v*) and then do the update of the weights at the end of each cycle. It means that we move the updateWeights(learningRate) function out of the traditional algorithm above. Instead of performing these operations for each record for each cycle, we reduce them to once per cycle. Therefore the total weights to be updated would be the summation of *dw* and *dv*, which are calculated by an individual pattern during the process. Note that the *dw*s are the weights to be added to the *w*, the weights between input layer and hidden layer. And *dv*s are the weights to be added to *v*, the weights between hidden layer and output layer. Here we assume there is one output neuron although more output neurons are possible. The updateWeights function can be done by binary reduction which can be performed in *logN* steps using *N* threads. Although the backwardPropagation and updateWeights functions may be parallelized, the gain of speedup may be limited. This setup allows us to assign one pattern to each thread in order to speed up the process.

Because the binary reduction operation may slow down the whole system performance, it is interesting to find out if assigning more than one pattern to a thread for processing will improve the performance. Once again, it depends on how many clock cycles will be used to synchronize the threads. If more threads are to be synchronized, then we would expect more time for the reduction operation. An interesting aspect is to find out how to organize operations so that we can get the best possible performance.

2.2 Moving Average

Moving averages can be used as financial indicators. There are various types of moving averages, of which two of the most popular are simple moving average and exponential moving average.

2.2.1 Simple Moving Average

Assume the daily closing price for day *t* is C_t. The *n*-day simple moving average can be defined as $SMA(t) = \frac{\sum_{i=t-1}^{i=t-n} C_i}{n}$, where C_i is the closing price at day *i*. So simple moving average can be computed by taking the average closing price of a stock, over the last *N* periods. Popular simple moving

averages are 5, 10, 20, 40, and 200. Let's assume that the last five periods for a stock are 1,3,5,7, and 9. Then, the 5 day simple moving average can be computed as (1+3+5+7+9)/5 = 5. While simple moving average giving all past *n*-day closing prices are weighted equally, the *n*-day exponential moving average assigns more weight to the most recent price, which will be discussed in the next subsection.

Simple moving average on a parallel computer can be done by using Prefix Sum operation. It also known as Scan operation. A Prefix Sum can be defined as the following. Given a set of *N* values $a_1, a_2, a_3, \ldots, a_n$, and an associative operation @, the Prefix Sums operation will compute the *N* quantities (a_1, $a_1@a_2$, $a_1@a_2@a_3$,..., $a_1@a_2@a_3@\ldots@a_n$). By using the Prefix Sum operation, the *N*-day Simple moving average of day *"i"* can be calculate as

$$SMA[i] = (prefixSum[i] - prefixSum[i-N])/N \quad (4)$$

where *SMA[i]* is the *N* day moving average at day *i*. Table 1 gives an example that uses Prefix Sum to compute 3-day moving averages. Assume the missing period (period -1, and -2 in our example) values are 0.

Period	1	2	3	4	5	6	7
Value	1	3	5	7	9	10	12
prefixSum	1	4	9	16	25	35	47
Total of subsequence	1	4	9	15(= 16-1)	21(= 25-4)	26(= 35-9)	31(= 47-16)
Simple Moving Average	1	2.5	3	5	7	8.67	10.33

Table 1 Example of calculation of *SMA* using prefixsum

2.2.2 Exponential Moving Average

Exponential moving average can be defined as $EMA(t) = (P_t - EMA(t-1)) * \alpha + EMA(t-1)$, where P_t is the closing price at day *t*. The α is weighting factor and can be defined as $\alpha = 2/(n+1)$, where *n* is the number time periods involved in the computation. For example, for 10-day period EMA, $\alpha = 2/(10+1) = 0.181818$, while 20-day EMA, $\alpha = 2/(20+1) = 0.095238$. One can rewrite the above EMA definition using past *n*-day closing prices and α as follows. Note that weights are decayed exponentially and the most recent prices carry more weights in the computation.

$$EMA(t) = \alpha P_t + \alpha(1-\alpha)P_{t-1} + \alpha(1-\alpha)^2 P_{t-2} \ldots + \alpha(1-\alpha)^{n-1} P_{t-n+1}$$

Even though *EMA* can be easily computed by using the method as mentioned in the previous paragraph, however it has serial nature in the computation. In order to deliver a parallel algorithm, let's define $\beta = (1-\alpha)$ where α is the multiplier that we define from the previous discussion. The formula to compute exponential moving average then becomes the following

$$EMA(t) = \alpha P_t + \alpha\beta P_{t-1} + \alpha\beta^2 P_{t-2} \ldots + \alpha\beta^{n-1} P_{t-n+1}$$

We will partition the whole data array into segments with lengths of powers of 2. Assume the leftmost data is the most recent data(the lowest index) and rightmost one (the highest index) is the oldest timing data. By using $O(N)$ memory to store the comupted information, we can therefore compute exponential moving average of P period in $O(log_2 P)$ time. Here, N is the number of records. Actually one can reduce the total memory to $2N$ as we will discuss shortly.

The computation of *EMA* involves two phases. In the first phase, we will generate the required data through iterations. In the first iteration, two pieces of data that are adjacent to each other will be processed to create a combined length-2 information. In the second iteration, combined length-4 segment information can be generated from the combined length-2 information. To simplify, let's assume p is power of two. In $log_2 P$ iterations, one can create $\frac{P}{2} + \frac{P}{4} + .. + 1$ which is a total of $P - 1$ pieces of data information for each segment of length *P*. This information will be used in the second phase of our *EMA* computation. Table 2 gives an illustration of the first phase *EMA* computation. Note that $\beta = (1-\alpha)$.

0	1	2	3	4	5	6	7
	(01)		(23)		(45)		(67)
		(0123)				(4567)	
				(1->7)			
	$c_0 + \beta c_1$	$c_0 + \beta c_1 + \beta^2(c_2 + \beta c_3)$	$c_2 + \beta c_3$	$c_0 + \beta c_1 + \beta^2(c_2 + \beta c_3) + \beta^4(c_4 + \beta c_5 + \beta^2(c_6 + \beta c_7))$	$c_4 + \beta c_5$	$c_4 + \beta c_5 + \beta^2(c_6 + \beta c_7)$	$c_6 + \beta c_7$

Table 2 Phase one of EMA Computation: data store scheme

For a segment of *P* periods of data on a *N* data array, where *P* < *N*, our job is to find the *EMA* for all the data on the data array with *N* data. Our approach is to partition any segment from index *a* to index *b*, where (*b-a+1*) is the period *P*, into at most two segments. It is possible that there is only one segment if the segment starts with index of power of 2 and ends with a power of 2 minus 1. In that case, we can compute the *EMA* value. It is just the value in the array of index $\lceil (a+b)/2.0 \rceil$. For example, if *a* is 56, *b* is 63, and *P* is 8, then index 60 holds the *EMA* value. Refer to Table 2 above to see how the combined values are stored. Note that this is the

best case since there is only one segment. For other *P-1* cases, out of every segment of length *P*, there are two segments. Let's assume these two segments are P_1 and P_2. First of all, we have to find out the cutline index. The first segment P_1 starts with index *a* and ends with an index which is to the power of 2 minus 1. Let's call this index *cutindex*. It can be computed as $b/p \times p$. Consider a segment from index 53 to 60: the resulting cutline index is 56. Note $60/8 \times 8 = 56$. The second segment P_2 starts with an index 56, which is to the power of 2, and ends with *b*, which is 60 in this case. Note that segment P_1 and P_2 can be formed by elements with lengths of power of two. For example, if segment $P_1$is of length 11, then it can be formed as a sum of segments of lengths 1, 2, and 8. Note here that the lengths of the segments from left to right are in increasing order. P_2 can be processed similarly, however the size of the component segments are decreasing. Distinguishing the order of increasing and descreasing of these components is important. For example, if P_2 is a segment of length 14, then the component sub-segments will have lengths of 8, 4, and 2. This is the order to retrieve the information. Note also that these component segments never carry the same length as other component segments in its combination. We can use a loop to mark the partioned segments if they are not zero and then add them up to get the resultant *EMA*. Note that during the loop computation, different power of β need to be applied to the retrieved value so the power of β shall be correctly applied to faithfully reflect the weights assigned to these segments. For the above example, a segment from 53 to 60 is considered. After computing the value of P_1, the resultant computing value from P_2 needs to be multiplied by β^3 before we add the resultant value from the computation of P_2. It is cubed because the segment 53 to 55 has length of 3. The multiplication by the power of β applies to the process of computing the values of P_1 and P_2 with the same reason. Details are omitted.

3 Conclusions

The experiments were conducted on a Intel i7 with Nvidia GeForce 550M. The parallelized moving average and neural network version was constructed using CUDA C. Due to the small size of records and the nature of the neural network, the speedup wasn't observed. The main reason is due to the requirement of synchronization of these threads. It is apparent that the expensive cost of synchronization makes the CPU implement more appealing. We expect that when the number of neurons and number of data increases, we shall get better results. We trust Nvidia will come up with a better solution to deal with the synchronization of threads.

For moving average computation, GPU computing does not provide advantages over CPU computing when there is a small amount of data. If the amount of data by simulation is increased, some improvement can be achieved. We understand that different machines with different models of CPU and GPU can create different results.

4 Conclusions

The parallel versions of moving average computation used in the financial industry and back propagation neural network computation were developed to run on Nvidia GPU using CUDA C. The GPU version of moving average does not give significant speedup over traditional CPU version using prefix sum operation. For neural network training process, GPU computation does not provide significant performance over traditional CPU implementation due to the requirement of thread synchronization. Even if we tried to minimize the number of synchronization by using device kernel calls, then speed up over the traditional CPU approach wouldn't be significant. Actually, in some situations when the number of records are small, the CPU implementation is superior. The implementation of real time predicting system over a huge data set in the financial industry is an interesting and challenging problem for future investigation.

5 References

[1] Myungho Lee, Chin Hong Chun, and Sugwon Hong, " Financial Derivatives Modeling Using GPU's", International Conference on Scalable Computing and Communications; The Eighth International Conference on Embedded Computing, pp 440 - 445

[2] Myungho Lee, Jin-hong Jeon*, Joonsuk Kim, and Joonhyun Song, " Scalable and Parallel Implementation of a Financial Application on a GPU: with focus on out-of-core case", 2010 10th IEEE International Conference on Computer and Information Technology, pp 1323 - 1327

[3] Feng Li, and Cheng Liu, " Application Study of BP Neural Network on Stock Market Prediction", 2009 Ninth International Conference on Hybrid Intelligent Systems, pp 174 - 178

[4] QianYu Lin, and ShaoRong Feng, " Stock market forecasting research based on Neural Network and Pattern Matching", 2010 International Conference on E-Business and E-Government, pp 1940 - 1943

[5] Nvidia, "Nvidia Cuda C Programming Guide", version 4.0, May 6, 2011,

[6] Ying Peng, Bin Gong, Hui Liu, and Bin Dai, "Option Pricing on the GPU with Backward Stochastic Differential Equation", 2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming, pp 19 - 23

[7] Steven Solomon, Ruppa K. Thulasiram and Parimala Thulasiraman, " Option Pricing on the GPU", 2010 12th IEEE International Conference on High Performance Computing and Communications, pp 289 - 296

[8] Edward Tirados and John Jenq, "Analysis of Leading Economic Indicator Data and Gross Domestic Product Data Using Neural Network Methods", Journal of Systemics, Cybernetics and Informatics, vol 7, no 4, 2009, pp 51-56

[9] Lei Wang, and Qiang Wang, " Stock market prediction using artificial neural networks based on HLP", 2011 Third International Conference on Intelligent Human-Machine Systems and Cybernetics, pp 116 - 119

[10] Yixin Zhou, and Jie Zhang, "Stock data analysis based on BP neural network", 2010 Second International Conference on Communication Software and Networks, pp 396 - 399

Exploration of Parallelization Frameworks for Computational Finance

Raj B Krishnamurthy[1], **Ikubin Chin**[2] **and Anjil Chinnapatlolla**[3]
[1]Exploratory Systems Lab, IBM Systems and Technology Group, Poughkeepsie, NY, USA
[2]Business Analytics and Optimization Lab, IBM Software Group, Tokyo, Japan
[3]System Software Lab, IBM Systems and Technology Group, Bangalore, India

Abstract *- This paper presents a comparison of parallelization frameworks for efficient execution of computational finance workloads. We use a Value-at-Risk (VaR) workload to evaluate OpenCL and OpenMP parallelization frameworks on multi-core CPUs as opposed to GPUs. In addition, we study the impact of SMT on performance using GCC (4.4) and IBM XLC (11.01) compilers for both single-precision and double-precision codes. We use an 8-core, 4-way SMT IBM Power7 with Linux (RHEL 6.0, 2.6.32 kernel) to evaluate OpenCL and OpenMP. Using the IBM XLC compiler, 2-way SMT is able to provide over 30% average improvement as compared to 1 SMT thread per core, whereas, 4-way SMT is able to provide over 50% average improvement as compared to 1 SMT thread per core.*

Keywords: *HPC, Supercomputing, OpenMP, Finance, Multicore, OpenCL*

1 Introduction

The global financial crisis of 2008 has pointed to the lack of efficacy of financial models and systems to run those models efficiently. Smarter systems can help bridge that gap by providing programming frameworks that are easy to learn, use and deploy. Also these frameworks must support languages that are functionally portable and provide the maximum degree of performance portability. As systems are confronted with the diminishing returns of clock-speed improvements, they must turn to uncovering task and data parallelism in workloads to maintain performance growth. This is needed to efficiently use the increasing multitude of cores that are being provided on modern CPUs today. Unfortunately, parallel programming is hard and significant research in academia and industry is attempting to bridge the gap between programmability, portability and performance [12].

OpenMP [2] has emerged as the de-facto standard for programming SMP (Shared-Memory) machines using user-provided compiler directives. The standard has also been extended recently to support accelerators and thread affinity [2]. OpenCL [1] (Open Compute Language) is vying to become the de-facto standard for programming accelerators. Originally designed for targeting graphics chips and accelerators, it is also emerging as a programming framework for parallelizing workloads on multi-core CPUs. In this paper, we investigate how OpenCL may be used for programming multi-core CPUs as *opposed* to GPUs (Graphics Processing Units). We compare the productivity and performance of OpenCL with OpenMP for multi-core CPUs. We also investigate how SMT (Simultaneous Multithreading) i.e. sharing of a CPU's physical resources by use of multiple hardware contexts can improve performance. This paper investigates the use of SMT for computational finance workloads that tend to be compute bound. In this paper, we use the term SMT "degree" to indicate the number of SMT threads per core.

Computational finance workloads are diverse in their computational characteristics. They tend to be sensitive to square root, exponential, logarithmic and reciprocal functions. They are run in single precision (SP) or double precision (DP) modes. A common practice is to replicate sequential implementations of these workloads across a large CPU cluster. In this paper we investigate how parallelism may benefit computational finance workloads. We built a VaR (Value-at-Risk) workload based on [3] and interaction with users at various conferences [13, 14, 18]. The VaR workload is rich with elementary math and transcendental math functions. It also displays significant amount of task and data parallelism.

We begin by exploring the characteristics of computational finance system stacks for computational finance in Section 2. Section 3 describes the parallelization frameworks that we evaluate in this paper. Section 4 presents the system architectures on which the aforementioned parallelization frameworks will be run. Section 5 describes the workload and avenues for parallelization in this workload. Section 6 describes implementation issues and Section 7 provides performance results with detailed analysis. Section 8 details the impact of performance evaluation. Section 9 discusses related work. Section 10 concludes the paper with future work.

2 Computational Finance Systems

We participated in several conferences over the past few years and learnt that customers in the HPC finance community use Linux for their computational needs [13, 14, 18]. The use of x86 CPUs (which support 2-way SMT) is highly widespread in this community. We decided not to take this practice for granted and investigate higher degrees of SMT cardinality or "degree". Users usually run models on R,

Matlab [15] or Mathematica [16]. They then convert this to C/C++ when the logic and mathematics become stable. The conversion to C/C++ allows programmers to check for numerical stability and convergence. There is a tendency to run these single threaded C/C++ programs by brute-force replication across large number of cores. The C/C++ code tends to be sensitive to sqrt(), log(), inv() and exp() performance.

We decided to explore parallelization so that users could get speedups on their codes, execute models in a reduced amount of time and be able to trigger changes to their business process because of this newly attained efficiency. We decided to investigate OpenCL [1] and OpenMP [2]. OpenMP has been widespread in the HPC community where automatic thread parallelization is sought by using compiler directives. OpenCL has become popular in the HPC community as the medium for programming GPUs. In this paper, we investigate how OpenCL can be used for parallelizing codes across *CPUs* especially when they are enabled for SMT.

Based on our user investigations and findings, we came up with the following requirements for systems that are suited for computational finance –

1) Run Linux on HPC building blocks

2) Support throughput computing using large number of threads or cores

3) CPU should also be capable of high single threaded performance

4) CPU, compilers and runtimes are optimized for sqrt(), log(), inv() and exp() performance. Additionally, they must support elementary math functions (e.g. IBM MASS [4] and linear algebra libraries (e.g. ESSL [4] and ATLAS)) with high performance

5) Provide parallelization that is user-directed and performance portable across a given set of platforms by investigating OpenCL and OpenMP

6) Provide the capability to port codes across new generations of hardware

The workload-optimized stack structure for computational finance is shown in Figure 1.

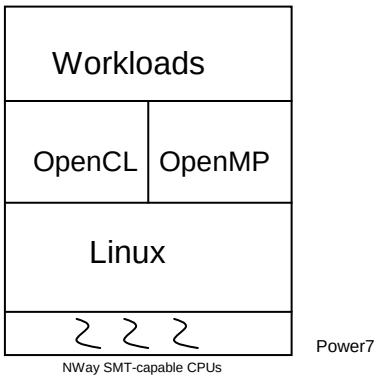


Figure 1. Initial Selection of Stack Components

3 Parallelization Frameworks

OpenCL (Open Compute Language) [1] is an open standard and is backed by a consortium of companies that has developed a framework for standardizing a high-performance computational language which is portable across a variety of platforms. Compute-intensive codes are packaged into a so-called OpenCL "kernel" that can be task- and data-parallelized. The OpenCL model is inspired from GPU programming (CUDA/OpenGL) and draws from the strengths of other parallel programming frameworks. The most fundamental action of an OpenCL kernel is to a "work item" and "work-items" are grouped into "workgroups". "Workgroups" are scheduled to hardware resources e.g. cores. The IBM OpenCL compiler is available on IBM developerWorks for POWER and x86 and this paper uses v0.3 [7]. The host CPU schedules kernels on the attached accelerator or CPU cores and must also ship data required by the kernels.

OpenMP [2] is a standard for parallel programming for shared memory machines and is backed by a consortium of companies. The user supplies OpenMP compiler directives to parallelize loops across resources. Operations like reduction are directly supported in the pragma. The standard is now being extended to support accelerators, thread affinity and embedded systems. IBM supports OpenMP pragmas in addition to its own pragmas for user-directed parallelization. We use OpenMP shipped with 2.6 Linux kernels for IBM Power7 for this paper. The OpenMP compiler generates pThread calls and pThreads are scheduled by the Operating System. We do not use any of the new OpenMP extensions for accelerators or thread affinity.

4 System Architecture

The system architecture used in this paper is a compute blade. This is essentially a multi-core system which is capable of large N-way SMT threads. We use the Power7 CPU [8] which has 8 cores and is capable of 4-way SMT for a total of 32 SMT threads. The CPU runs Linux (RHEL 6.0 with a 2.6.32 kernel). Both OpenCL and OpenMP may be parallelized across multiple CPUs. SMT threads are exposed

by the Operating System as additional "logical" CPUs. OpenCL kernels and OpenMP threads are scheduled across multiple SMT threads. We use GCC 4.4 and IBM XLC 11.01 for C/C++ code compilation. IBM XLCL v0.3 is used for OpenCL kernel compilation. We use the MASS [4] library (Mathematical Acceleration Subsystem) for optimized elementary and transcendental math functions with our C/C++ code. This is used as a high-performance alternative to the standard math library. In the next section, we describe the computational finance workload used to evaluate our system architecture.

5 Risk Analysis Workload

This section describes the risk analysis workload and its computational composition. We also highlight workload components where parallelism may be uncovered. The risk analysis workload uses the "value-at-risk" financial framework which will be described next. The workload has been built with user input and is based on the mathematics described in [3].

5.1 VaR – Value-at-Risk

Value-at-Risk is the potential loss in value of a risky asset over a defined period of time for a given confidence interval. VaR is widely used in commercial and investment banks to capture the potential loss in a traded portfolio from adverse market movements for a given period, e.g. If the VaR of an asset is $ 100 million at one-week, 95% confidence interval, it means that there is a 5% chance that the value of the asset will drop more than $100 million over any given period of one week. There are several popular computational methods used to calculate VaR – (i) historical simulation, (ii) variance-covariance, (iii) Monte-carlo and (iv) Delta-gamma. This paper uses the Monte-Carlo method and will be described next.

5.2 Monte-Carlo VaR

A Monte-Carlo method is used to approximate the probability of outcomes by performing so-called "random walks". "Random walks" simulate multiple trials of a random variable. In the case of VaR, portfolio returns over a given period of days is computed and then sorted. The largest loss corresponding to the VaR period is reported as shown in Figure 2. There are number of ways of computing portfolio returns over a given period but we use the Black-Scholes model to price financial instruments. In this model, the price of a financial instrument consists of log(), inv(), sqrt() and exp() functions in its closed-form representation.

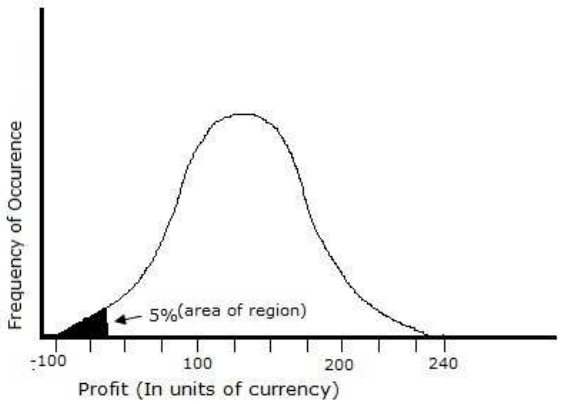


Figure 2. Value-At-Risk Distribution

5.3 Monte-Carlo VaR Parallelization

The Monte-Carlo VaR implementation pseudo code is below. As a first step, we provide the co-variance matrix which captures the portfolio details, the number of portfolios, number of Monte-Carlo simulations and the confidence interval. A number of parallelization strategies are possible. We could equally divide the portfolios across multiple cores and each of the portfolio groups could be computed concurrently. Alternatively, a single portfolio could be computed at a time and the simulations could be parallelized across multiple cores. We use the latter strategy. The values needed for random walks are computed using random number generation which can be computationally intensive. Each simulation prices a financial instrument using a closed form representation composed of log(), inv(), sqrt() and exp() functions. These returns (in currency units) are sorted and the partial distribution is sent to the root CPU for aggregation and merge-sorting. After all CPUs complete their computation, the root CPU forms a distribution similar to Figure 4. The VaR calculation picks the loss corresponding to the confidence interval from the sorted distribution, which is aggregated from all participating CPUs.

1. Supply covariance matrix, number of portfolios, number of simulations and confidence interval
2. For portfolio = 1…N
3. **do**
    1. Compute random numbers 1..M
    2. For simulations = 1 … M
    3. **do**

        Compute portfolio return by pricing financial instruments in the Black-Scholes Model

    End Loop

4. Sort and form partial distribution of returns vs frequency

5. Send partial distribution to root CPU

6. Root CPU Merge-Sorts partial distributions

7. Root CPU forms aggregate distribution

8. Root CPU extracts VaR value from aggregate distribution

End Loop

6 Implementation Issues

This section describes the implementation issues for OpenCL and OpenMP code. We used the Armadillo [17] library for implementing the VaR code, since it gave us function wrappers for BLAS and LAPACK libraries. In addition, it gave us matrix template classes. We used the cholesky decomposition and convolution operations as the principal linear algebra functions. Our elementary math functions included the erlang function, log(), inv(), sqrt(), rnd() and exp().

6.1 OpenCL

We encoded the entire Monte-Carlo simulation into an OpenCL kernel. We support both single precision and double precision calculations inside the kernel as a compile-time option. We did not pre-calculate random numbers but supply them to the kernel during execution. The IBM OpenCL runtime v0.3 reported upto 32 workgroups for the Power7 blade with 32 SMT threads across 8 cores with 4-way SMT. It also reported a global workgroup size of 1024. We inferred that a single workgroup is being scheduled to each SMT thread and used a local workgroup size of 32. The OpenCL program evaluates a single portfolio at a time and the simulations for a particular portfolio are mapped across the local workgroup size of 32 work-items in a given workgroup.

6.2 OpenMP

For the OpenMP implementation, we compute a single portfolio at a time and parallelize the simulations across multiple SMT threads of the 8-way Power7 system. A single OpenMP pragma was needed to parallelize the simulations.

7 Performance Evaluation

We now describe the performance evaluation of the risk analysis workload for OpenCL and OpenMP implementations. We measure speedup and execution time against the number of SMT threads available on the system. We have an 8-way Power7 blade which is capable of supporting 32 SMT threads. The Risk Analysis Workload (VaR) is evaluated with a portfolio size of 10,000 and with 112,640 simlulations. We measured each data-point three times and used the average in the plots below to eliminate any OS-jitter effects. We configured the Linux OS scheduler to schedule user-level threads on SMT hardware contexts in an even fashion so load is evenly balanced, e.g 11 threads across 8 cores means that 1 SMT thread is scheduled on each core, while 3 cores have an additional SMT thread running. Each user-level software thread is mapped to a single SMT thread.

7.1 Single Precision Floating Point (SP)

Figure 3 shows the execution-time performance of OpenMP (with standard math library, also called OpenMP-std-math), OpenCL and OpenMP-with-MASS implementations using the GCC compiler with –O3. MASS is the IBM Mathematical Acceleration Subsystem [4] library which is optimized for transcendental and elementary math operations. The OpenCL compiler (v0.3, XLCL) was used to compile the OpenCL kernel and then GCC was used to link and bind all object modules. The performance of the code with the standard math library performs worse than the OpenCL and OpenMP-with-MASS implementations.

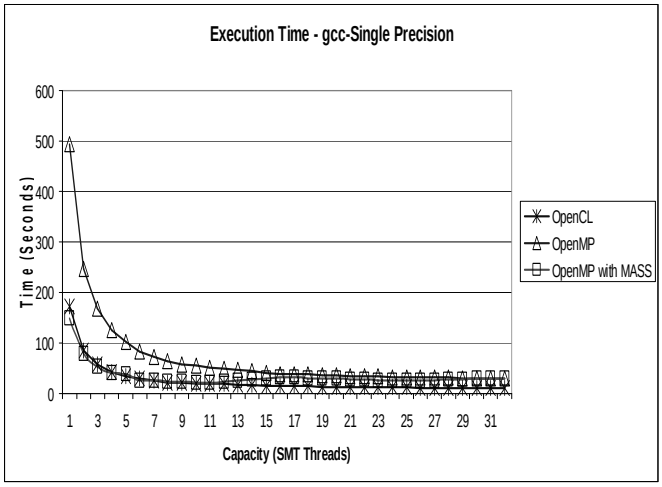


Figure 3. Execution Time vs SMT Threads (GCC, SP)

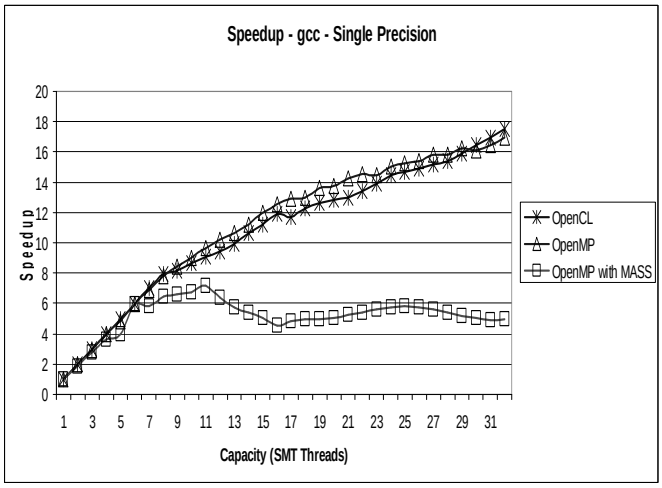


Figure 4. Speedup vs SMT Threads (GCC, SP)

The OpenMP-with-MASS implementation tracks the OpenCL compiler performance until 11 SMT threads and then starts diverging. The OS scheduler schedules 8 SMT threads across 8 cores and 3 threads across 3 cores, so 3 cores have 2 SMT

threads running. We ran the "perf stat" profiler and found that IPC decreases and cache misses of the GCC-generated code increases beyond 11 SMT threads. We attribute this to resource conflicts due to OpenMP thread data sharing using the highly-optimized MASS library. Enhanced MASS optimizations leads to reduced number of pipeline stalls and cache misses. Thus, they do not benefit from SMT. Note that OpenMP with the standard math library does not display this behavior as it does not provide efficient optimizations. OpenCL code compilation follows a two-step process and the IBM Power OpenCL compiler is able to do a better job of compacting thread and data state to avoid resource conflicts. 4-way SMT on Power7 for this workload is able to provide a 50% reduction in elapsed time at 32 SMT threads (four per core) over 8 SMT threads (1 per core) for the OpenCL and OpenMP-std-math codes.

The speedup curves in Figure 4 show OpenMP-with-std-math and OpenCL codes providing near-linear speedups but the OpenMP-with-MASS codes flattening after 11 SMT threads. This is easily attributed to enhanced optimization with MASS leading to higher floating point pipeline utilization that does not benefit by SMT. Additionally, the single thread performance of OpenMP-with-MASS is higher than OpenMP-with-std-math and slightly higher than OpenCL.

We used the IBM XLC compiler to compile our codes and found that the OpenMP-with-MASS implementation does much better than the corresponding GCC-generated code of Figure 3. In fact, the OpenMP-with-MASS implementation equals or betters the OpenCL implementation using the IBM XLC compiler. The execution times are shown in Figure 5 and the speedups are shown in Figure 6. The speedup curves of OpenCL are close to linear than OpenMP-with-MASS because the OpenMP implementation does better at execution time of 1 SMT thread but not at higher thread counts, as the enhanced optimization leads to optimal resource utilization of the floating-point pipeline and cache.

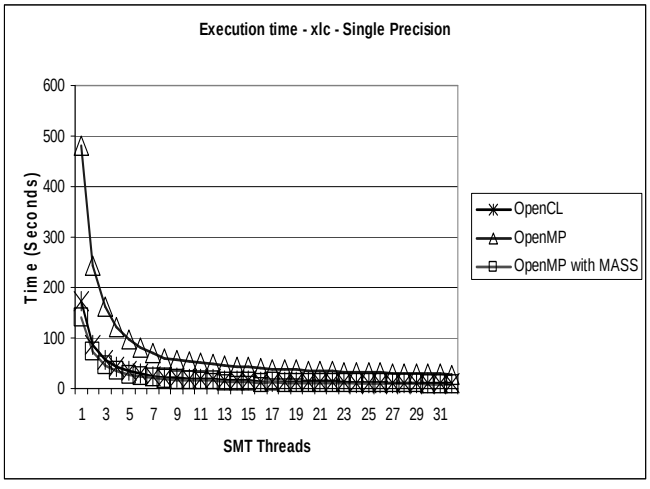


Figure 5. Execution Time vs SMT Threads (XLC, SP)

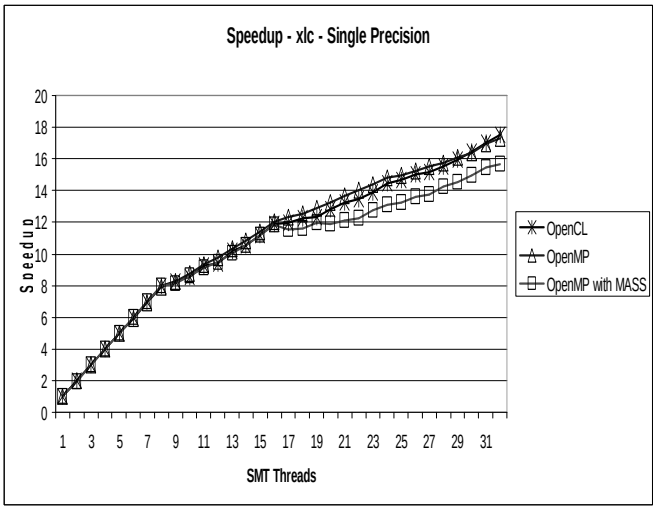


Figure 6. Speedup vs SMT Threads (XLC, SP)

7.2 Double Precision (DP)

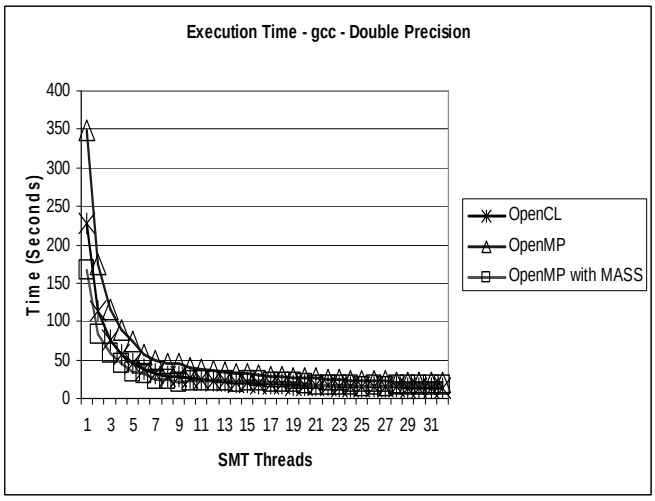


Figure 7. Execution Time vs SMT Threads (GCC, DP)

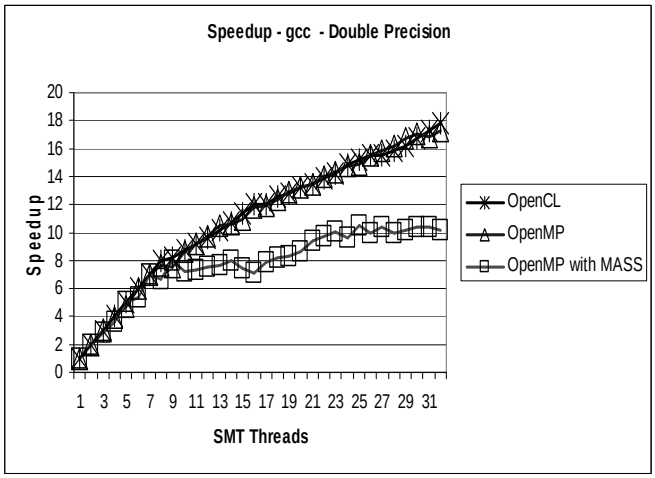


Figure 8. Speedup vs SMT Threads (GCC, DP)

Most codes in computational finance are run with double precision floating point and we discuss the results with GCC- and XLC-generated code. Figure 7 and Figure 8 show GCC-generated code performance for OpenCL, OpenMP-std-math and OpenMP-with-MASS. The OpenMP-with-MASS code shows similar behavior as seen for the single precision case where the execution time elongates slightly at 12 SMT threads due to resource conflicts. In Figure 8, OpenCL and OpenMP-with-std-math show close to linear performance but OpenMP-with-MASS flattens after 8-10 SMT threads. This is attributed to the higher performance of OpenMP-with-MASS at 1 SMT thread. Additionally, the use of MASS leads to efficient code that utilizes floating point and cache resources efficiently, leading to poor speedups when physical cores are shared with SMT.

Figures 9 and 10 show the performance of XLC-generated code. Both OpenMP-std-math and OpenMP-with-MASS performance exceeds the performance of the OpenCL code using the IBM XL compiler. This is attributed to better scheduling of code and resources in double precision mode by the XLC compiler leading to better utilization of the floating-point pipeline. If Figure 9 is compared with Figure 5, where Figure 5 is XLC-generated code in single precision mode, OpenMP-with-std-math in double precision does better than corresponding single precision code. In comparison, XLC-generated OpenCL code in double precision does worse than single precision code. We attribute the better performance of OpenMP-with-std-math code in double precision to better utilization of the floating point pipeline and instruction scheduling by the XLC compiler. In the case of OpenCL and OpenMP-with-MASS, both see reduced performance for DP code as the CPU can usually sustain more SP operation throughput than DP operation throughput. For DP code, OpenMP-with-MASS does much better than OpenCL as the XL compiler for C/C++ is able to do a better job of instruction scheduling and floating-point utilization than the XL compiler for OpenCL.

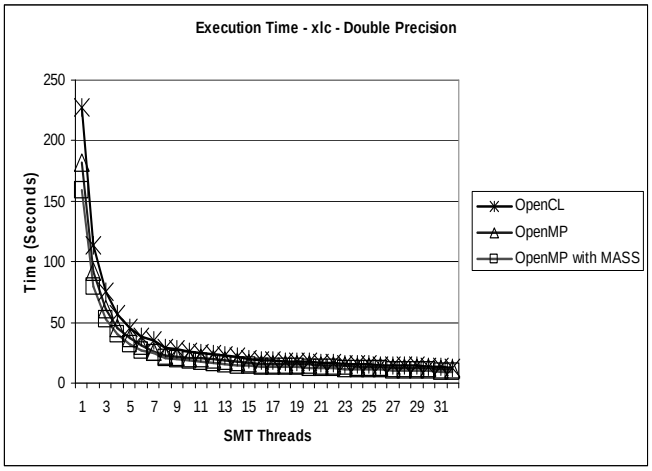


Figure 9. Execution Time vs SMT Threads (XLC, DP)

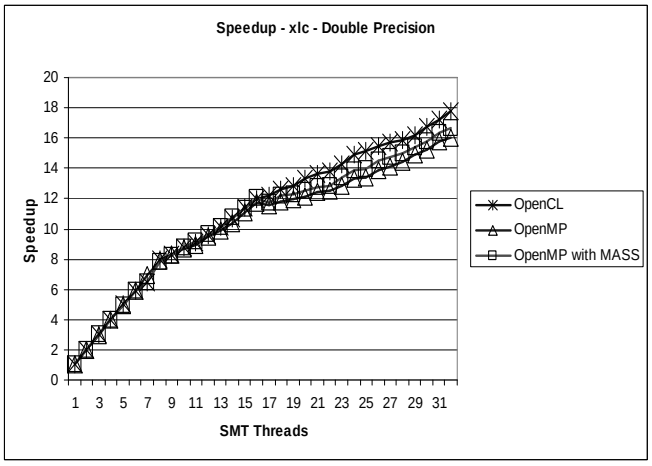


Figure 10. Speedup vs SMT Threads (XLC, DP)

The speedup curves of Figure 10 show OpenCL code showing close to linear performance while the OpenMP code flattening around the 16 SMT thread region. This region involves two SMT threads sharing a physical core. The sublinear performance in this region is easily attributed to resource and cache conflicts.

7.3 Impact of SMT

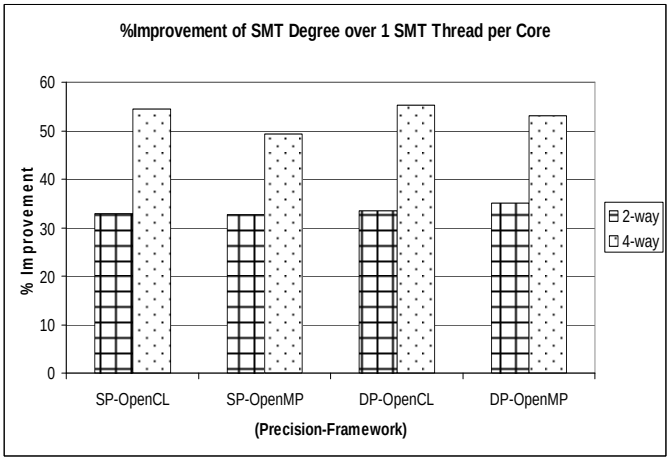


Figure 11. Impact of SMT Degree (XLC)

Figure 11 shows the impact of SMT degree for the VaR workload using the XLC compiler. 2-way SMT gives an average of 33.4% improvement across OpenCL and OpenMP while 4-way SMT gives an improvement of 53% across OpenCL and OpenMP. For the 2-way improvement calculation, 16 SMT threads on the 8-core system (2 threads/core) are compared to 8 SMT threads (1 thread/core). For the 4-way improvement calculation, 32 SMT threads on the 8-core system (4 threads per core) are compared to 8 SMT threads (1 thread/core). All comparisons are based on execution times of the VaR (Value-at-Risk) workload for different SMT thread counts using the XLC compiler.

8 Post-Evaluation Analysis

For the homogeneous multi-core architecture, the learning curve for OpenCL can be steep. For OpenMP, we simply had to use a single-line OpenMP pragma to parallelize the entire program, although, we could only match OpenCL performance using the right choice of IBM compiler toolchains and libraries. With OpenCL, we can run the OpenCL kernel unchanged from a CPU to an accelerator like a GPU. OpenCL allows portable interface with accelerators but OpenMP provides ease of use with high-level OpenMP directives. We were able to match OpenMP and OpenCL performance using the right set of compiler tool chains and libraries. The stack of Figure 1 should be updated to reflect the use of IBM XL C/C++ compilers and MASS libraries. For programming homogeneous multicores, OpenMP with XLC and MASS seems to emerge as a formidable framework. It provides better productivity and can match OpenCL performance.

OpenCL provides a portable way to program multicores and accelerators. For example, one may prototype programs on a multicore and then launch these on CPU-GPU clusters. OpenCL is a low-level mechanism to exact parallelism and performance. OpenMP relies on the compiler and so the right compiler tool chain and libraries is needed to match OpenCL performance. OpenACC [5] is a newly emerging standard that stipulates compiler directives for use with accelerators and provides best of both worlds – OpenMP and OpenCL. The results of this paper reveal that 4-way SMT provided by the POWER architecture can lead to additional benefits by reducing execution time. In other words, if application code optimizations and compiler action are sub-optimal, SMT degree can be used to get better processor pipeline utilization and enhanced execution time. SMT is beneficial for the VaR workload because SMT threads are able to share instructions and data.

9 Related Work

[9, 10] focus on random number generation and their implementation complexity on the IBM Cell BE accelerator for value-at-risk calculations. They do not focus on understanding scaling issues and implications of programming models and frameworks. Similarly, [11] focuses on market risk calculations on GPUs (Graphics Processing Units). They also do not focus on scaling or use of parallelization frameworks. This paper uses CPUs for market risk calculations and investigates the use of SMT to increase processor utilization, while analyzing scaling.

10 Conclusion and Future Work

We evaluate OpenCL and OpenMP for both single precision and double precision VaR codes. We find that although OpenCL has a steeper learning curve, it provides good performance without use of additional libraries. OpenMP is based on user-directed parallelism but requires the right compiler toolchain (XLC) and library combination (MASS) to match OpenCL performance. The use of OpenCL and OpenMP parallelization frameworks can lead to better utilization of POWER cores using SMT. We find that SMT degree does help execution. We are currently evaluating a library that will allow OpenCL kernels to be dispatched onto attached blades from a large SMP system. This will allow computation to be scheduled close to data than moving data to the computation. We are also addressing how OpenCL kernels can be accelerated by explicit use of data parallelism. OpenCL allows portability of kernels across CPUs and accelerators. The new OpenMP directives [2] are also attempting to make OpenMP easy to program accelerators. OpenACC [5] is also headed in the direction of using compiler directives to program accelerators. As the race to find "Utopia" for parallelization frameworks continues, the systems community must constantly evaluate workloads against new system architectures. This will help determine which workload classes benefit from new and emerging parallelization frameworks.

11 References

[1] http://www.khronos.org/opencl/

[2] http://openmp.org/wp/

[3] Benninga, S and Wiener, Z. Value-at-Risk (VaR). Mathematica in Education and Research. Vol. 7, No. 4, 1998.

[4] http://www-01.ibm.com/software/awdtools/mass/linux/

[5] http://www.openacc-standard.org/

[6] Black, Fischer; Myron Scholes (1973). "The Pricing of Options and Corporate Liabilities". Journal of Political Economy 81 (3): 637–654. doi:10.1086/260062.

[7] https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=80367538-d04a-47cb-9463-428643140bf1

[8] http://www-03.ibm.com/systems/power/advantages/power.html

[9] V. Agarwal, L.-K. Liu, and D.A. Bader, ``Financial Modeling on the Cell Broadband Engine," *22nd IEEE International Parallel and Distributed Processing Symposium* (IPDPS), Miami, FL, April 14-18, 2008

[10] D.A. Bader, `` On the Design of Fast Pseudo-Random Number Generators for the Cell Broadband Engine and an Application to Risk Analysis," *37th* International Conference on Parallel Processing, Portland, OR, Sept 8-12, 2008

[11] Matthew Dixon, Jike Chong, and Kurt Keutzer. 2009. Acceleration of market value-at-risk estimation. In *Proceedings of the 2nd Workshop on High Performance Computational Finance* (WHPCF '09).

[12] Krste Asanovic, "The Landscape of Parallel Computing Research: A View from Berkeley", Technical Report No. UCB/EECS-2006-183, University of California at Berkeley, Berkeley, California.

[13] http://www.flaggmgmt.com/hpc/

[14] http://www.flaggmgmt.com/linux/

[15] http://www.mathworks.com/products/matlab/

[16] http://www.wolfram.com/mathematica/

[17] http://arma.sourceforge.net/ (C++ Linear Algebra Library)

[18] www.sifma.org/ (Securities and Financial Markets Association)

Computational Finance with Map-Reduce in Scala

Ron Coleman, Udaya Ghattamaneni, Mark Logan, and Alan Labouseur
Computer Science Department
Marist College
Poughkeepsie, NY, 12601

Abstract – This paper presets results of computational finance experiments using map-reduce in Scala. We observe superlinear speedup, super-efficiency, and evidence for a high degree of compute and I/O overlap in the median runtimes using "naïve," memory-bound, fine-grain, and course-grain parallel algorithms on three different hardware platforms.

Keywords - Computational finance, map-reduce, Scala, actors, parallel programming, bond pricing, securities pricing

1 Introduction

Computational finance is a multidisciplinary field at the crossroads of mathematical finance and computer science. [29] The emphasis is on development and utilization of numerically intensive methods for pricing, risk analysis, forecasting, automated trading, and other applications. Map-reduce [3] is a framework generally to speed-up data analysis using distributed computing. While map-reduce has been applied to different problem domains, many of a data-intensive nature, almost no attention has been given to opportunities for computational finance as a mixture of floating point and data-intensive operations.

Scala [23] is a modern, high-level Java Virtual Machine (JVM) language that blends object-oriented and functional programming styles with actors, a shared-nothing model of concurrent computation inspired by physics theories. [13] Proponents have argued that Scala language features are suited to solving large-scale computing tasks on inexpensive, commodity multicore and multiprocessor platforms [11] in an expressive manner that avoids the concurrency hazards and run-time inefficiencies of shared, mutable state programs. Indeed, the function-oriented style of Scala would seem to lend itself precisely to coding mathematical expressions which characterize quantitative operations.

We endeavored to put Scala's scalability claims to the test on a large quantitative problem, namely, finding the fair value of bond portfolios. We chose bonds because fixed income pricing theory is fairly transparent. [6, 14, 29] Furthermore, the price of a bond informs or is closely related to a number of different financial instruments, including annuities, bond derivatives, and interest rate swaps, the most heavily traded financial derivative in the world. [14] Thus, we posit that the practical performance implications of bond portfolio analysis extend to scientific and technical computing.

In this paper, we study several parallel algorithms in relation to two serial algorithms to price bond portfolios using map-reduce in Scala. For three different multicore platforms, the data generally show superlinear speedup and super-efficiency [9] in the median runtime that is in line with the maximum performance expectations published by the original equipment manufacturer (OEM) [15]. The data show that the parallel "naïve" algorithm, which assigns one portfolio per map function object, has median performance little different from, if not better than, memory-bound, fine-grain, and coarse-grain parallel algorithms. In other words, the Scala actor model, which underlies map-reduce, appears to exploit hardware threads (i.e., "hyper-threads") very efficiently. To our knowledge these findings have not be reported elsewhere.

The Scala source code we used in this study and detailed experimental results are available online [1,7].

2 Related work

The literature shows enduring interest in speeding up computational finance algorithms [2,22,26], most notably using specialized hardware and ad hoc low-level programming techniques [19,22], which do not, in our experience and opinion as programmers, promote software reusability or programmer productivity. The literature furthermore indicates map-reduce is a widely accepted approach to speeding up computation for various problem classes, although many of these efforts appear to be of a data-intensive nature. [3,8,16,27] Few, to our knowledge, have investigated computational finance applications. Zhang, et al [32] investigates map-reduce for option pricing using the backward stochastic differential equation (BSDE) method. However, the focus there is on the computational aspect of solving BSDE; we consider the end-to-end process, namely,

compute and I/O phases that characterize more practical settings. Woo and Xu [31] use map-reduce to perform market basket analysis with Amazon's EC2 service. This effort is geared toward data analysis, not computational finance; and cloud computing rather than parallel processing (i.e., speedup and efficiency) which is our primary focus. There is at least one cloud map-reduce framework for JVM languages [10], although this framework has have not been specifically designed for Scala. We instead use a pure-Scala map-reduce API from Haller and Sommers (H-S) [12]. Scala also has a library which is not actor-based in which it is also possible to employ map and reduce parallel functions directly on parallel-enabled collections (e.g., lists, arrays, etc.). [25] We do not investigate parallel collections here.

3 Methods

In this section we develop the methods of the paper. We discuss of bond pricing theory, portfolio valuation, the serial processing of portfolios in Scala, and the parallel processing of portfolios using map-reduce in Scala.

3.1 Bond pricing theory

A bond is a fixed income security. [6, 14, 29] For our purposes in this paper, we are considering only simple bonds, b_i, defined by the 5-tuple:

$$b_i = [i, C, n, T, M] \tag{1}$$

i is an integer type which plays no part in bond pricing except to uniquely identify the bond in an inventory which we describe below; C is the coupon payment amount; n is payment frequency per annum; T is the time to maturity in years; and M is the face value due at maturity. The sum of the net present value of these cash flows, C and M, is the price of the bond. Thus, the fair value, $P(b_{i,}r)$, of a bond, b_i, is functionally defined follows:

$$P(b_i, r) = \sum_{t=1}^{n \times T} \frac{C}{(1+r_t)^{t/n}} + \frac{M}{(1+r_T)^T} \tag{2}$$

The parameter, r, is the time-dependent yield or yield curve, the general discussion of which is beyond the scope of this paper. For simplicity sake without loss of generality, we use the United States Treasury on-the-run bond yield curve and interpolate between the tenors (i.e., bond maturity dates) using polynomial curve fitting.

It is worth noting that Equation 2 informs a number of different quantitative finance calculations which is one of the reasons we have chosen to model bonds. The first term of the right hand side of Equation 2 is an annuity embedded in the bond's price. The bond price also is used to calculate the conversion factor for bond futures and the fair value of bond forwards and bond options. Swap contracts, the most heavily traded financial derivative in the world [14], involve discounted cash flows similar to Equation 2. Indeed, the bond price itself may also be used in swap valuation. The point is that Equation 2 underlies and plays a part in a number of other quantitative finance problems and for this reason bond pricing is important as a kind benchmark computation.

A portfolio is a collection instruments, in our case, bonds. The fair value, $P(\phi_j)$, of a portfolio, ϕ_j, with a basket of Q bonds is functionally defined as follows:

$$P(\phi_j) = \sum_{q=1}^{Q} P(b_{\phi(j,q)}, r) \tag{3}$$

In other words, the portfolio's value is the sum over the value of all its constituent bonds.

3.2 Bond generation algorithm

We generate simple bonds that model a wide range of computational scenarios. The goals are to 1) produce a sufficient number of bonds to mimic realistic fixed income portfolios and 2) avoid biases in commercial-grade bonds that depend on prevailing market conditions. Specifically, we have the collections, $\vec{n}$={1, 4, 12, 52, 365), $\vec{T}$={1, 2, 3, 4, 5, 7, 10, 30}, and $\vec{\delta}$={0.005, 0.01, 0.02, 0.03, 0.04, 0.05}. We derive the parameters for a bond object from the bond generator equations below:

$$M=1000 \tag{4a}$$

$$n = \vec{n}[\bullet] \tag{4b}$$

$$T = \vec{T}[\bullet] \tag{4c}$$

$$C = \mathrm{M} / \mathrm{T} \times \delta[\bullet] \tag{4d}$$

where $\bullet$ is an integer uniform random deviate in the range of [0, s); and s is the size of the respective collection. We invoke Equations 4a - 4d a total of 5,000 times to produce the bond inventory, V, which we store in an indexed document-oriented database that we describe later.

We generate a portfolio by first selecting its size, that this, the number of bonds, Q, as per the equation below.

$$Q = v + \sigma \times \eta \tag{5}$$

η is a Gaussian deviate with mean of zero and one standard deviation. σ and v are configurable parameters set to 60 and 20, respectively. Finally, we construct a

basket of size, Q, bonds for a portfolio, ϕ_j , per the equation below.

$$i = \bullet \quad (6)$$

where $\bullet$ is an integer uniform random deviate in the range of $[1,|V|]$ and $|V|$ is the size of the bond inventory. The universe, U, of bond portfolios are also stored in a persistent repository.

3.3 IO design

We store the inputs, the bond and portfolio objects, b_i and ϕ_j in MongoDB, a document-oriented database. [20] We also store the outputs, the portfolio prices, in the database. A portfolio document, ϕ_j, does not contain the bond, b_i, directly but a reference to the bond, namely, the bond's primary key, i. In other words, the database is in third object normal form [17,19]. This means that to evaluate Equation 3 end-to-end, 2+Q accesses are required: one to fetch ϕ_j, Q to retrieve each b_i, and finally one to update the portfolio with the price.

3.4 Pricing algorithms

All the pricing algorithms, serial and parallel, work with the database in the same way. They retrieve from the database the same subset, $U' \subseteq U$, of random portfolios and then retrieve the bonds that have been randomly assigned to a portfolio. The algorithms evaluate Equation 2 and Equation 3, and then update the database with the result of Equation 3.

3.5 Serial algorithms

There are two serial algorithms. Both algorithms price portfolios serially using a triple-nested `foldLeft` function, the inner `foldLeft` for Equation 2, the middle `foldLeft` for Equationn 3, and the outer `foldLeft` passes over all portfolios in *U'*. One serial algorithm fetches a bond from the database only when the bond is needed by the pricing function literal. The other serial algorithm, the "memory-bound" algorithm, works the same way except it pre-fetches into memory all the bonds that are needed by the portfolios.

3.6 Parallel naïve algorithm

The naïve parallel algorithm is so-called because it "naively" assigns one portfolio to a mapping function whose inner `foldLeft` evaluates Equation 2 and whose outer `foldLeft` evaluates Equation 3. The H-S function, *mapreduceBasic* method, relates portfolios in *U'* to two functions: a mapping function and a reducing function. The prototype for our mapping function is below.

```
def mapping(portfId: Int,
                    r: List[Double]):
  List[(Int,Result)]
```

The parameter, *portfId*, is the portfolio id and the parameter, *r*, is the time-dependent yield curve. The return result is a list with one element, a two-tuple that has the portfolio id and the pricing result. The prototype for the reducing function is below

```
def reducing(portfId: Int,
                 vals: List[Result]):
  List[Result]
```

Note that this function is trivial: it just needs to return the only element in the parameter, *vals*, list.

3.7 Parallel memory-bound algorithm

The parallel memory-bound algorithm is nearly identical to the naïve algorithm. The difference is that the bonds are first pre-loaded into memory from the database, just like the serial memory-bound algorithm except the transfer of bonds to memory is done in parallel with actors and task-level parallelism. However, the parallel memory-bound algorithm uses task-level parallelism using actors to load the bonds in parallel. The parallel memory-bound reducing function which H-S *mapreduceBasic* requires is the same as the naïve reducing function. The parallel memory-bound mapping function prototype is below.

```
def mapping(portfId: Int,
              bonds: List[SimpleBond]):
  List[(Int, Result)]
```

3.8 Parallel coarse-grain algorithm

The parallel coarse-grain algorithm uses the same mapping and reducing functions as the naïve algorithm. The difference is the coarse-grain algorithm uses the H-S *coarseMapReduce* method which takes two extra parameters: the maximum number of mappers and the maximum number of reducers. We set the number of mappers and the number of reducers both to the number of processors which we get from the Java class, *Runtime*. The *getRuntime().availableProcessors()* method of this class returns the number of hyper-threads, not the number of cores or execution units. [4] As far as we know, it is not possible to get the number of actual hardware execution units except from the OEM data sheets.

3.9 Parallel fine-grain algorithm

The parallel fine-grain algorithm is similar to the naïve algorithm. The difference in this case is its map function uses a single (non-nested) `foldLeft` to evaluate

Equation 2. The reduce function evaluates Equation 3. Thus, the prototype for the map function we pass to the H-S *mapreduceBasic* method is below.

```
def mapping(portfId: Int,
            bondId: Int):
  List[(Int,Result)]
```

The *bondId* parameter is the bond id which the mapping function uses to retrieve the bond object. However, since bond id cannot be known in advance, it corresponding portfolio was pre-loaded prior to invoking the mapping function. This pre-load time is taken into account in the runtime measurement. The portfId parameter is only used to link the bond price result with its portfolio which allows the reducer to reduce the bond prices properly.

The reducing function sums all the bond prices by their portfolio id. The reducing function prototype is the same as the naïve algorithm.

4 Experimental design

In this section we describe the experimental design, including the hardware environment, trials, and speedup calculations.

4.1 Environment

The environment consisted of three hardware platforms of different Intel multicore processors. The table below gives the system configurations and year of introduction in each case.

Table 1. Platforms

CPU	Clock	Cores	Threads	RAM	Year
W3540	2.93	4	8	4 GB	2009
i5-650	3.20	2	4	4 GB	2010
T3300	2.50	2	2	3 GB	2008

The clock is in giga-hertz and the year is the year the processor was introduced by Intel. The W3540 and i5 systems run Microsoft Windows 7 Professional, Service Pack 1. The T3800 runs Microsoft Windows XP, Service Pack 3. The code is compiled by Eclipse 3.7 [5] using the Scala IDE plugin [28] version 2.0.0 with the 64-bit JVM on the W3540 and i5 and the 32-bit JVM on the T8300.

4.2 Trials

To obtain statistically conservative results [8], we use the median result of 11 to 22 trials in which we observe the runtime performance for an ensemble of workloads of *U'* portfolios randomly selected portfolios from a total of *U*=100,000 portfolios. Any given workload in the ensemble contains $u=2^x$ portfolios where $x=\{0,1,2,3\ldots10\}$ for each trial. Since the mean number of bonds in each portfolio is 60 with a standard deviation of 20. The largest workload analyzes *u*=1024 portfolios with an average of about 60,000 bonds total.

We measured the serial algorithm time, T_1, for processing an ensemble using each of the platforms listed in Table 1 above. We also measured the map-reduce parallel algorithm time, T_N, for naive, coarse-grain, and fine-grain map-reduce algorithms. In the case of the memory-bound algorithms, we have given the compute and the I/O runtimes separately. The memory-bound compute measurement is the median runtime to value the portfolios. The memory-bound I/O measurement is the median runtime to load the bonds into memory.

To get a better understanding of how IO might be impacting the process, we furthermore separated out compute and IO phases by first loading all the bonds into memory before processing them in both serial and map-reduce cases.

4.3 Speed-up calculations

Given T_1 and T_N , we have the speed-up, *R*.

$$R = T_1 / T_N \tag{7}$$

N is going to be the number of cores or execution units, not the number of hyper-threads.

With *R* we have, *e*, the efficiency.

$$e = R / N \tag{8}$$

5 Results

We run trials from *u*=1..1,024 bond portfolios.

5.1 Parallel naïve results

The results for the naïve (∗), memory-bound compute (+), and memory-bound I/O-only (△) measurements for the W3540 are summarized in the graph below.

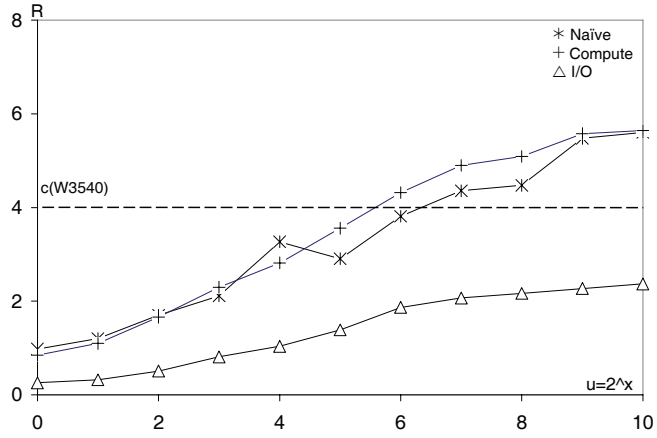


Figure 1. W3540 results

Note that speedup, R, above the dashed line is superlinear and super-efficient performance. The corresponding numerical data for the W3540 for

u=1,024 bond portfolios for the naive algorithm is in the table below.

Table 1. W3540 with u=1,024 bond portfolios

	T_N	R	e
Naive	8.58	5.64	141%
Memory-bound compute	7.13	5.60	140%
Memory-bound I/O	2.50	2.37	59%

The naive algorithm results for the i5 are summarized in the graph below.

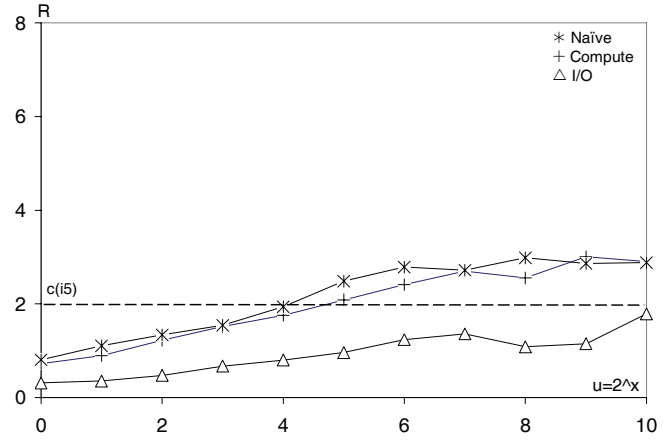


Figure 2. i5 results

The corresponding numerical data for the i5 for u=1,024 bond portfolios for the naive algorithm is in the table below.

Table 2. i5 with u=1,024 bond portfolios

	T_N	R	e
Naive	16.87	2.90	145%
Memory-bound compute	13.42	2.88	144%
Memory-bound I/O	3.54	1.79	89%

The naive algorithm results for the T8300 are summarized in the graph below.

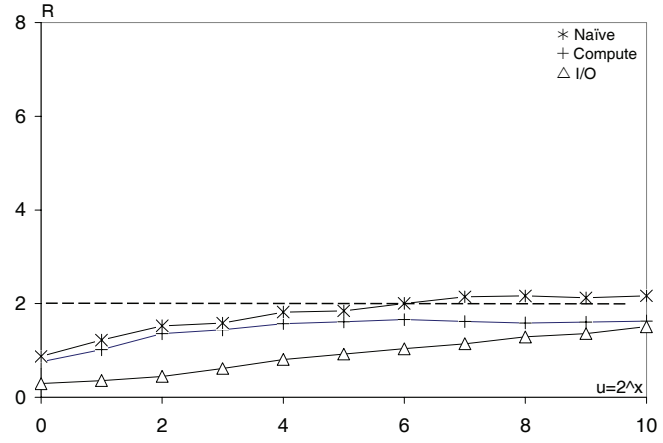


Figure 3. T8300 results

The corresponding numerical data for the T8300 for u=1,024 bond portfolios for the naïve algorithm is in the table below.

Table 3. T8300 with u=1,024 bond portfolios

	T_N	R	e
Naive	73.77	1.63	81%
Memory-bound compute	67.34	2.16	108%
Memory-bound I/O	8.07	1.51	75%

5.2 Parallel fine-grain results

The composite fine-grain results for the W3540 (×), i5-650 (□) , and 8300 (◇) are summarized in the graph below.

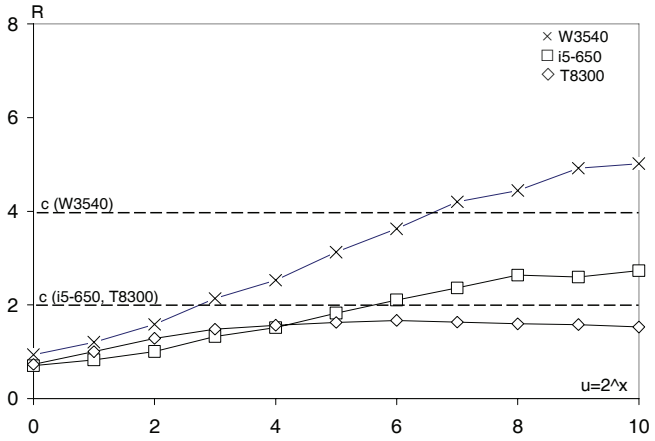


Figure 4. Fine-grain results

The corresponding numerical data for the W3540, i5-650, and T8300 for 1,024 bond portfolios for the naive algorithm is in the table below.

Table 4. Fine-grain with u=1,024 bond portfolios

	T_N	R	e
W3540	9.65	5.02	125%
i5-650	17.93	2.73	137%
T8300	78.52	1.53	76%

5.3 Parallel coarse-grain results

The coarse-grain results for the W3540, i5, and T9300 are summarized in the graph below.

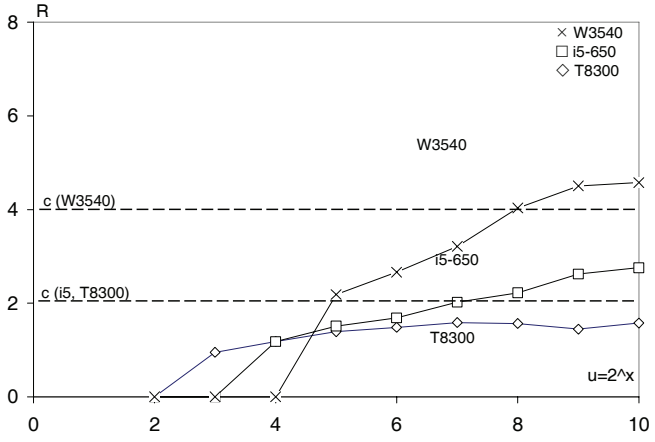


Figure 5. Coarse-grain results

The corresponding numerical data for the W3540, i5-650, and T9300 for 1,024 bond portfolios for the naive algorithm is in the table below.

Table 5.

	T_N	R	e
W3540	9.16	5.29	132%
i5-650	19.55	2.50	125%
T8300	79.20	1.51	76%

6 Discussion

The naïve algorithm appears to be the best performing overall end-to-end, achieving super-linearity and super-efficiency for levels of u, depending on the processor type. For instance, the more modern processors, the W3540 and i5, realize super-linearity and super-efficiency for u as small as 64.

Most interestingly, I/O is broadly sub-linear which, by itself, is not surprising. However, I/O does not appear to be a processing bottleneck since the difference between compute and memory-bound compute plus memory-bound I/O over the range of u appears to be insignificant.

7 Conclusion

This investigation of map-reduce for computational finance lead us to consider other avenues for future work. First, we would like to explore changes to H-S to support multiprocessor parallelism. Second, there are open questions on how to "shard" or parallelize the data. Finally, we had briefly mentioned Scala's parallel collections. A very worthwhile study might examine parallel collections and compare the coding style and performance to H-S map-reduce.

8 Acknowledgements

This research has been funded in part by grants from the National Science Foundation, Academic Research Infrastructure award number 0963365 and Major Research Instrumentation award number 1125520.

9 References

[1] http://code.google.com/p/scaly/ retrieved on 30 January 2012

[2] Crosman, P., "Bloomberg Uses GPs to Speed Up Bond Pricing," *Wall Street Technology*, 24 September 2009

[3] Dean, J. and Ghemawat, S. "MapReduce: Simplified Data Processing on Large Clusters," OSDI 2004

[4] http://docs.oracle.com/javase/6/docs/api/java/lang/Runtime.html#availableProcessors() retrieved on 29 January 2012

[5] http://eclipse.org/ retrieved on 29 January 2012

[6] Fabozzi, F.J., Mann, S.V., *Introduction to Fixed Income Analytics*, 2nd ed. Wiley, 2010

[7] http://foxweb.marist.edu/users/ron.coleman/ retrieved on 30 January 2012

[8] Georges, A., et al, "Statistically Rigorous Java Performance Evaluation," *OOPSLA '07*, October 21-25, Montreal, Quebec, Canada

[9] Gustafson, J.L., "Fixed Time, Tiered Memory, and Superlinear Speedup," *Proceedings of the Fifth Distributed Memory Computing Conference*, 8-12 Apr 1990, p.1255-1260

[10] http://hadoop.apache.org/ retrieved on 29 January 2012

[11] Hill, M.D., Marty, M.R., "Amdahl's Law in the Multicore Era," IEEE Computer Society, July 2008

[12] Haller, P., Sommers, F., *Actors in Scala*, Artima, 2011

[13] Hewitt, C.; Bishop, P., and Steiger, R., "A Universal Modular Actor Formalism for Artificial Intelligence", *IJCAI*, 1973

[14] Hull, J., *Options, Futures, and Other Derivatives*, Prentice Hall, 1989

[15] "Hyper-Threading Technology," *Intel Technology Journal*, Vol. 6, Issue 01, 14 February 2002

[16] Lammal, R., "Google's MapReduce Programming Model Revisited," http://www.cs.vu.nl/~ralf/MapReduce/paper.pdf retrieved on 11 Mar 2012

[17] Lodhi, F., Mehdi, H., "Normalization of object-oriented design," *Seventh International Multi-Topic Conference*, Lahore Pakistan, 8-9 Dec 2003, p. 446-450

[18] Ma, Z., Gu, L., "The Limitation of MapReduce: A Probing Case and a Lightweight Solution," *Proc. of the 1st Intl. Conf. on Cloud Computing*, Nov. 21-26, 2010, Lisbon, Portugal

[19] Merunka, V., et al, "Normalization Rules of the Object-Oriented Data Model," *EOMAS '09* Proceedings of the International Workshop on Enterprises & Organizational Modeling and Simulation

[20] http://www.mongodb.org/ retrieved on 30 January 2012

[21] NVIDIA, "Computational Finance on the GPU," *GPU Technology Conference*, San Jose, CA, 30 Septemter – 2 October 2009

[22] NVIDIA, "JP Morgan Speeds Risk Calculations with NVIDA GPUs", *HPC Wire*, 4 August 2011

[23] Odersky, M., et al, *Programming in Scala*, Artima, 2008

[24] Owens, J.D., et al, "GPU Computing," *Proceedings of the IEEE*, Vol. 96, No. 5, May 2008

[25] Prokopec, A. et al, "A Generic Parallel Collection Framework," EPFL, *InfoScience* 2011, 2010-7-31

[26] Schmerken, I., "Wall Street Accelerates Options Analysis with GPU Technology," *Wall Street Technology*, 11 Mar 2009

[27] Ranger, C., et al, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," *HPCA*, 2007

[28] http://scala-ide.org/, retrieved on 29 January 2012

[29] Tuckman, B., *Fixed Income Securities*, 2nd ed., Wiley Finance, 2002

[30] Tsang, E.P.K., Martinez-Jaramillio, "Computational Finance," *IEEE Computational Intelligence Society*, August 2004

[31] Woo, J., Xu, Y., "Market Basket Analysis Algorithm with Map/Reduce of Cloud Computing," *2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, WORLDCOMP'11 July 18-21, 2011, Las Vegas Nevada, USA

[32] Zhang, Y., et al, "Parallel option pricing with BSDEs method on MapReduce," *Third International Conference on Computer Research and Development* (ICCRD), Shanghai,11-13 Mar 2011, p289-293

SESSION

PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS

Chair(s)

Prof. Hamid R. Arabnia

Multicore Clusters for CFD Simulations Comparative Study of Three CFD-Softwares

A. de Blanche, N. Namaki, S. Mankefors-Christiernin
Division of Automation and Computer Engineering
University West, Trollhättan, Sweden

Abstract

Multicore processors have come to stay, fulfill Moore's law and might very well revolutionize the computer industry. However, we are now in a transitional period before the new programming models, numerical algorithms and general computer architecture have been developed and the software has been rewritten. This paper focuses on the effects multicore based systems have on industrial computational fluid dynamics (CFD) simulations. The most significant finding was that five of the models ran faster when only one process was executed on each multicore node instead of two. In these cases the execution time was increased by between 6.5% and 64% with a median increase of 10% when utilizing both cores.

Keywords: multicore, cluster, non-uniform inter-process communication

1 Introduction

Getting the most out of any computer system, and especially a new architecture, has always been a challenging task. When new hardware technologies are introduced and put to use it is not obvious that the application performance will increase – even if the technology is immensely superior. The existing applications have been optimized to run on the current systems and they might not at all be suitable for the next-generation high-performance computers. The latest major addition is the introduction of multicore systems.

As suggested by James Peery et. al. [1], at Sandia national laboratory, the execution time might increase if more cores are utilized. Using Sandia key algorithms they showed that when using sixteen cores the performance was barely on par with two cores. According to Arun Rodrigues this is an issue to which the industry has no known solution and that often is ignored [1]. The addition of a second core could theoretically double the computing capacity of a computer node [2], however in many cases the bottleneck moved to another component. As pointed out by [3] [4] [5] the most notable is the processor versus memory size and bus ratio, both when it comes to latency as well as bandwidth.

Traditionally high performance computer designs have tried to balance three factors [6]: Compute power, memory and I/O-capacity. According to Vaughan et. al. [7] the new generation of COTS distributed parallel computers (clusters) [8] [9] has added inter-node communication capacity as a crucial factor. For Multicore processors Vaughan's addition should be split into two, namely intra-node and inter-node communication capacity due to the non-uniform inter-process communication architecture.

In this paper we present a, first study on the effects multicore based systems have on industrial computational fluid dynamics (CFD) simulations. Do multicore processors facilitate a decrease in execution time within the CFD area or will the already long execution times increase? During our investigation we found that for five of our nine models there were configurations in which the execution times increased by 6.5% to 63% when utilizing two cores in each processor instead of one. Six configurations paged to the hard drive due to lack of memory and were excluded.

2 Multicore Architecture and Issues

Typical modern computer architecture is based on a microprocessor, system memory, busses and various other components. Very much like the schematic overview in figure 1. All information that goes from the main memory to the processor has to pass through the system bus (Front Side Bus/FSB), the memory controller and the memory bus.

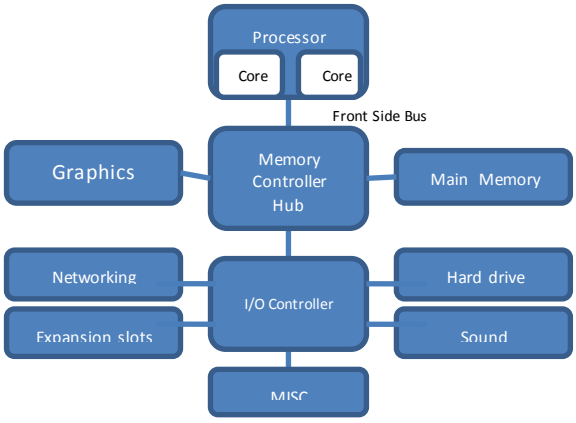


Figure 1: Architectural overview of a modern computer

In this section we go through the four most important systems in a computer; processor, memory, I/O-capacity and inter-process capacity. The three first are proposed by [6] with the addition of *inter-node communication* suggested by [7], although we find inter-process capacity to be a better description. Over the last 30 years the execution speed of a microprocessor has increased from 5 MHz to approximately 4 GHz. During this time many new techniques that improves the processors were introduced, such as pipelining and speculative execution as well as advanced instruction level parallelism. The old recipe of creating a faster processor by raising the clock frequency relied on the possibility to

manufacture smaller components inside the processors. Advances in this field are no longer possible due to the laws of physics.

The solution to higher speeds taken by almost all manufacturers is multiple processor cores on the same chip.

2.1.Memory Challenge

So far the memory architecture shown in figure 1 has not changed notably between the single and multicore processors. One of the major challenges of the multicore systems will be to design a system that can keep up with the tremendous computing power in a single multicore chip. This puts enormous pressure on, among other things, the FSB and the memory subsystem. At the moment the memory subsystem, on its own, is not fast enough to keep up with a single core processor; as a result it utilizes a two to four level memory hierarchy of increasingly faster memories closer to the processor. When extra cores are added some memory problems might be fixed by adding additional memory levels [3].

By comparing the historical capacity of processors using the SPEC-benchmark [10] with the FSB and memory bus [10] [11] we find that the difference has increased quite drastically over the years. Consider an application that was written for a system in 1993 and perfectly optimized for that "balance", would on a modern computer wait for memory approximately 50% of the time. It is a crude comparison and it is based on using one core in the modern multi core chip.

The problem of sustaining a high memory throughput to several processors is not a new problem for multiprocessor systems [12]. However, there is a difference between using multiple processors and multiple cores.

2.2.Inter Process Communication Challenge

All but the simplest parallel and distributed applications implement some level of inter-process communication. One problem that arises with the introduction of multicore clusters is that they have a non-uniform inter-process communication architecture. There are at least two different levels of communication costs involved when passing messages in a multicore cluster, either over the loopback interface or over the network.

Consider an application with four processes that implements message passing. If designed in 2003 it would be written to work well on Pentium 4 processors and gigabit Ethernet.

When executing a 16 process job on 4 computers each equipped with 2 dual core processors. The inter-process communication will be even more non-uniform with the additional two different levels of intra-node communications. Especially when compared to the 16 single core computers that the application was designed to be executed over. In that case the link capacity would still be $1/16^{th}$ of gigabit Ethernet, since it scales linearly.

2.3. I/O Utilization

The fourth important subsystem in regards to system balance is the I/O-subsystem. However the CFD applications that we focus in this investigation are very light on the I/O side. They only read the input files in the beginning of the execution and write the output at the end. Therefore we do not have any focus on the I/O utilization.

3 Computing Platform and Benchmark Applications

The aim of this experiment is to investigate the effect multicore processors have on applications in the Computational Fluid Dynamics (CFD) domain. Or more precisely, to determine if the efficiency of a simulation increases when two simulation processes are executed on each dual core node compared to one process per node.

3.1. High Performance Computing Platform

The Intel Core 2 is a common processor in many high performance environments. The cluster used in this investigation consists of 180 identical computers based on the Core 2 E6550 processor. The E6550 is manufactured using 65nm technology and has 291 million transistors.

The Intel Core 2 processor has two processor cores, each core has 2*32KB of private L1 cache (data and instructions), at the next level the cores share a 4MB L2 cache. Having a shared L2 cache has its benefits as well as drawbacks. While it allow cache sharing between cores and allowing the entire cache to be used by one of the cores as well as decreasing the cache impact of moving processes between cores it make the cores compete for the cache resources the hit latency is longer than for separate caches [12].

The memory controller and front side bus (FSB) of the Core 2 system are located off-chip, i.e. the architecture mimics that in figure 1. The transfer rate of the FSB has a maximum capacity of 10.6 gigabyte/s while the memory sustains 5.33 gigabyte/s per channel. The upper limit for fetching data from memory is limited to 10.6 gigabyte/s. However, for this to be achieved there has to be no other transfers affecting the busses. If the data is saved on the circuits in the same memory bank, the maximum theoretical transfer speed is 5.33 gigabyte/s.

The STREAM benchmark [13] [14] was used to measure the sustainable bandwidth of the memory system for reference purposes. During several executions with different sized matrixes of a size between 48 and 1800 megabyte, the sustained bandwidth never exceeded 4080 megabyte/s. When the matrix size was increased to span both memory channels the total sustained bandwidth was even decreased to 3900 megabyte/s.

The cluster is connected by a hierarchy of gigabit switches. During the experiments the computers were connected by a single gigabit Ethernet switch. A more detailed description of the hardware can be found in table 1.

Table 1: Cluster computer and network equipment specifications.

Processor	Intel Core 2, E6550
Clock Frequency	2.33GHz
# of cores	2
Hyper threading	No
Technology	65nm
Transistors	291 Millions
L1 Cache	32 KB code and 32KB data
L2 Cache	4MB cache shared between cores, non-

	inclusive with L1 cache
Front side bus	Q35
Frequency	1333MT/s
Bandwidth	10.6 gigabyte/s
Memory	DDR2
Size	2GB
Specification	PC2-5300
# of channels	2, dual channel
Frequency	667Mhz
Bandwidth	2 * 5.33 gigabyte/s
Measured Bandwidth	4.08 gigabyte/s (STREAM [13])

3.2.Applications and Models Used

Three applications were used in this investigation. Two commercial CFD applications; CFX [15] and Fluent [16] and the open source package Openfoam [17]. Nine CFD models, three per application, were used in the experiments. Since the behavior of these applications are dependent of the type of flow simulated so are the solution times. Three representative "real world" models were selected from two multinational high-tech companies for each of the two commercial applications. The CFX models ith Openfoam one real world and two research models from scientists were used together with Openfoam. Table 2 gives an overview of the models which have bearing on the aerospace domain.

Table 2: Sizes and names of the nine models used in the evaluation. The Number of elements gives a first order approximation of the size of the model.

Name	CFX	Fluent	Openfoam
M1	817 460	817 460	702 000
M2	1 811 036	743 223	1 980 900
M3	2 935 295	5 529 223	5 379 520

The sizes of the models as well as the run-time differ between models. As an example the Fluent M1 require approximately 4 hours to execute all the way through over eight nodes while Fluent model 2 require somewhere around 12 hours to complete and the M3 models need several days.

Those knowledgeable in the area of computational fluid dynamics know that there are two basic ways to determine when a simulation should end, either by specifying the number of calculations (iterations) to run or by specifying some convergence criteria (transient) [18].

When dealing with real world industrial applications and models one problem is that they are not well defined small units that execute in a minute. Real world CFD calculations are huge problems that often require tens of gigabytes of memory and take weeks to execute on a single computer. If all models in this investigation were to be executed all the way though it would require several years of runtime. In [19], Yang, et. al. however showed that a partly executed simulation fairly well matches a full execution of the code. Their empirical performance prediction results are constrained to cases where the problem and input sizes do not vary.

To be able to perform this investigation we decided to limit the execution time of the experiments. It was decided that we would use the iteration method (mentioned above) to limit the execution time. Each model was executed, in parallel, over eight computer nodes, with one process per node. After 1000 seconds of execution, the executing iteration was extracted from the application log file. This iteration was then used as the stop criteria for all experiments with that particular model, ensuring the exact same number of iterations and execution behavior in all comparable runs for that model-application combination. In this way only the changes in the number of processes and hardware could be responsible for the changes in run-time.

4 Experiment Setup

To determine the usefulness of multicore processors within the engineering simulation domain, the nine (9) fluid simulations in table 2 and their corresponding applications were executed in two different setups, Single and Dual.

Although utilizing the same hardware the difference between the two setups was that in the single setup each process is executed on one dual core node, as if it were a single core system, although the second core will execute various other system tasks. The dual setup on the other hand refers to when two processes are placed on each dual core computer node.

This means that when executing a eight processes job the Single setup will use eight dual core nodes and place one process on each node, effectively only utilizing one of the cores for simulation purposes. The dual setup, however, would employ four computers and utilize both cores in all machines: using a total of eight cores.

The models (table 2) were pre-partitioned into [2 4 6 8 16 32] partitions to be run in parallel by the same number of processes. This partitioning was done using the applications' default mechanisms.

For the simulations where the amount of memory needed for the execution was greater than the available system memory (figure 3 and table 6), thus forcing the systems to page to disk, was removed from the study. The reason behind this was that their execution times increase by several orders of magnitude. This threshold was reached earlier with the Dual setup since the node memory is shared between two processes. When adding more computer nodes and processes the amount of global memory effectively increases which in turn allow the applications to run entirely in memory.

During the execution of the applications the run time was measured and the processor, memory and network loads were monitored. To calculate the runtime a timestamp were made before and after the execution of the pre-partitioned simulations. The timestamps were taken with the gettimeofday() system call which has a relatively high resolution (ms). Although compared with a runtime between 253 and 6388 seconds the error is negligible.

The processor, memory and network loads were calculated as averages over 180 second intervals. These measurements were all made through the /proc kernel interface. One measurement set consumed ~8 ms of processor time, according to measurements done by strace [20].

5 Results

In order to investigate the effect multicore processors have on our three applications and the nine models that were executed in a

Single and Dual core setups, see section 4. Figure 3 a-i) contain graphs depicting the results from the experiments. The black lines show the execution time of the Single setup, i.e. where one process is placed on each node, and the dashed line represents the Dual setup where two processes are placed on each dual core node.

At a first glance the results look as one would expect them too, the Single setup for the same number of processes out performs the Dual setup, see figure 4. This was expected since the Single setup use two times the number of computers and effectively has twice the amount of memory, twice the level 2 cache (not applicable to all multicore processors) and an extra core to use for running operating system and other tasks, etc.

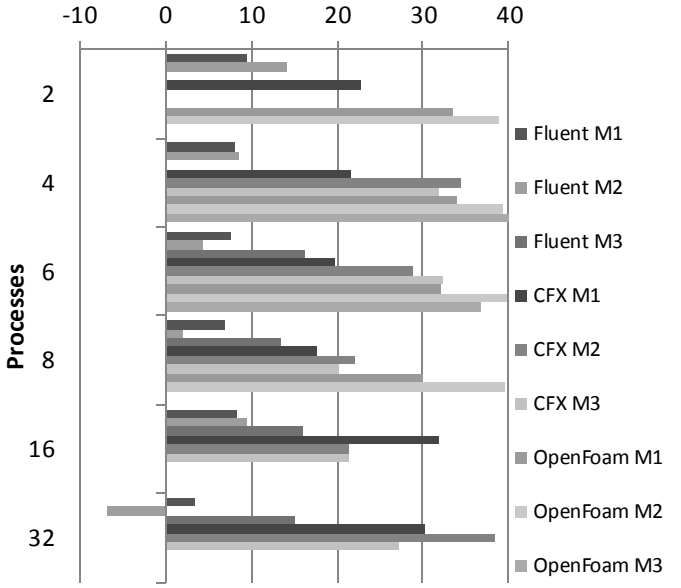


Figure 4: The number of percent by which the Single setup outperforms the Dual setup for the same number of processes.

This was the case for all models, apart from Fluent model 2, when divided into 32 processes as can be seen in figure 3 b). The dual setup outperformed the single with 22 seconds out of 341, so by ~6.5%. Fluent model 1 is not much larger and shows the same behavior, although the single setup still outperforms the dual with 3.5%, at 32 processes.

In table 4 the speedups (scalability) of the different applications and models are listed. To compare both single and dual setups with each other the speedup of the dual setup uses the same baseline as the single. Thus all values for each model are normalized after the single setup with the least amount of processes.

The speedup obtained when adding processes to the computation does not follow the theoretically best possible of $\mathbf{T_s/P}$ (time to execute simulation/number of processes) but for software in the CFD area scalability in this range is considered comparatively well [18] [21].

In table 3 we can see that when going from two Single to two Dual processes, i.e. from two cores on different computers to two on the same, Openfoam does not perform as good as the other two applications. Openfoam M2 models also shows a strange plateau, at 4 to 6 nodes as well as 8 to 16, where there is little to none scalability. Although for eight nodes it shows a decent scalability. Fluent M3 shows a similar, but not at all as bad, pattern between eight and sixteen nodes.

Quoting the execution times and speedups based on the amount of processes the problem is divided into is a straightforward approach. However, when dealing with computer clusters and multicore processors the speedups and execution times depending on the amount of computers is of interest.

Table 3: The execution time (T) of the different setups normalized to the equivalent lowest number single setup execution, T_N. The norm was given the value 100.

Model	Setup	Processes					
		2	4	6	8	16	32
Fluent M1	Single	100	53	39	32	17	13
	Dual	110	58	42	34	19	13
Fluent M2	Single	100	57	48	34	22	18
	Dual	116	62	50	35	24	17
Fluent M3	Single	-	100	72	63	57	24
	Dual	-	-	86	73	68	28
CFX M1	Single	100	54	39	34	22	17
	Dual	130	69	49	41	32	24
CFX M2	Single	100	56	40	34	21	15
	Dual	-	85	57	44	27	24
CFX M3	Single	100	55	41	35	23	18
	Dual	-	80	60	44	29	24
OFoam M1	Single	100	47	31	24	-	-
	Dual	150	72	46	34	18	-
OFoam M2	Single	100	52	53	26	-	-
	Dual	163	85	88	44	43	-
OFoam M3	Single	-	100	93	59	-	-
	Dual	-	269	147	59	41	-

5.1.Dual core Processor Efficiency

Turning to the less obvious result, for some models/setups the problem is actually solved faster when only one process instead of two is executed on each dual core computer node. This occurred at the higher number of computers and the speedup was between 6.5% and 64%, see table 4 for details. For the lower number of computer nodes the Dual setup always outperform the Single – given that there is enough resources to execute at all. E.g. the Fluent M2 model required more than 4 gigabyte of ram and would there for not execute on 2 machines or less. Then as more nodes are added the time gap between the Single and Dual setups close and for 5 of the 9 models the curves actually switch place, leaving the Single setup to outperform the Dual for the higher number of nodes.

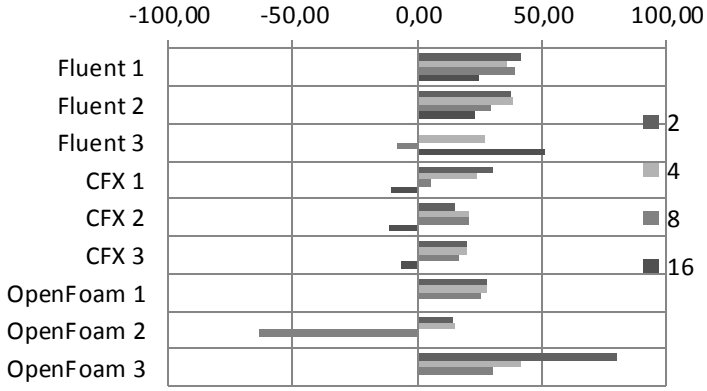


Figure 5: The number of percent by which the dual setup outperformed the single setup for the same number of computers [2,4,8,16].

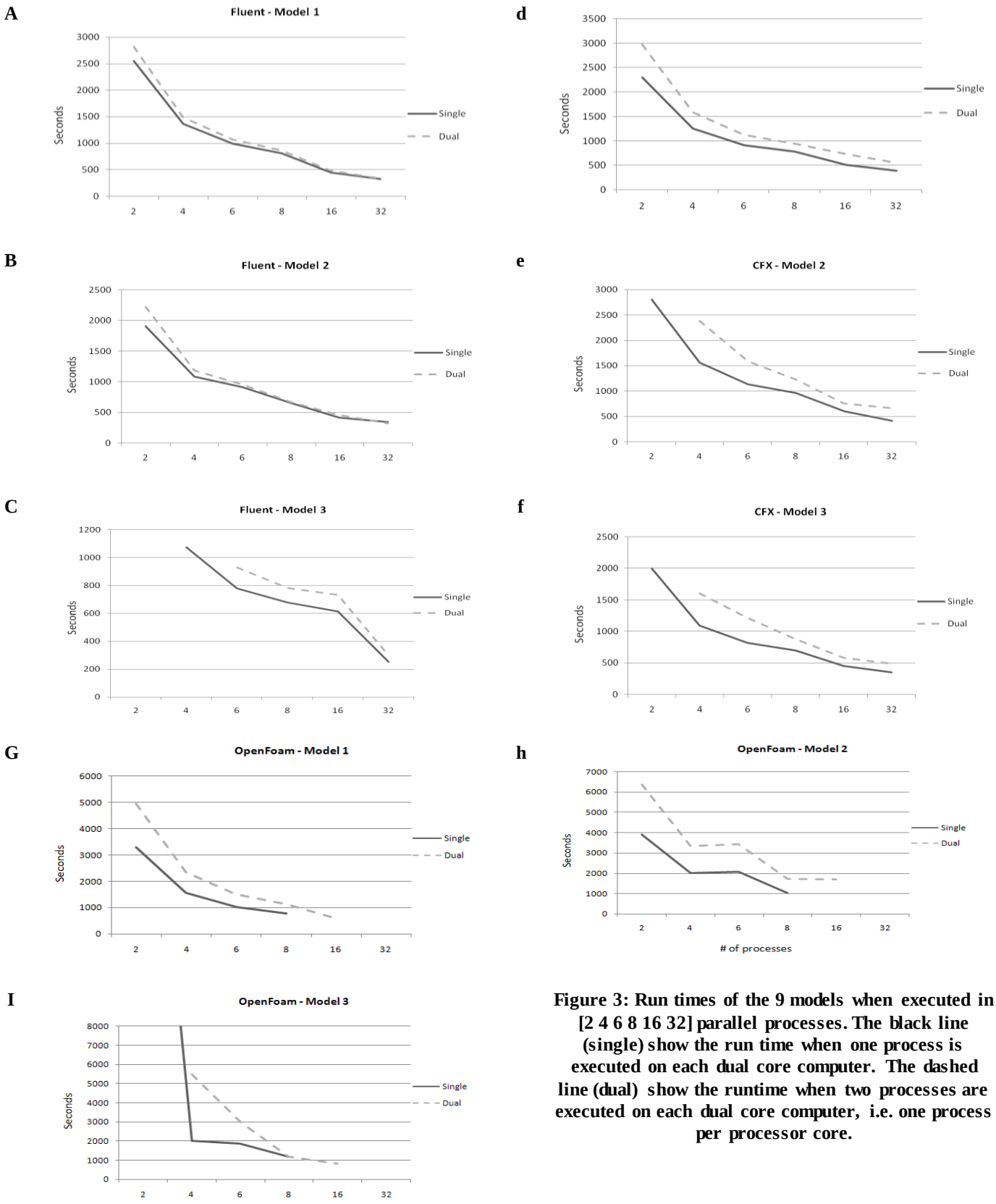


Figure 3: Run times of the 9 models when executed in [2 4 6 8 16 32] parallel processes. The black line (single) show the run time when one process is executed on each dual core computer. The dashed line (dual) show the runtime when two processes are executed on each dual core computer, i.e. one process per processor core.

The Single node executions that outperformed their Dual counterparts are listed in table 4 together with the speedup gained. Using Fluent model 3 as an example, the application actually solves the problem faster if you divide it into 8 parts and run it on 8 dual core processors compared to dividing it into 16 parts and running one process on each core on 8 dual core nodes. Hence, if there are 8 dual core nodes it will take 8% longer time to execute the problem if you utilize both cores instead of just one. The same applies to all CFX models when executed on 16 computers as well as for Openfoam model M2 when executed on 8 computer nodes. For the Openfoam model M2 the speedup gained, when executing one process per computer instead of two was as high as 63%.

Table 4: The following models execute faster when executed with one process on each dual core computer than when executed with two processes on each computer.

Model	# of computers	Speedup for single
Fluent M3	8	8%
CFX M1	16	10%
CFX M2	16	11%
CFX M3	16	6,5%
OF M2	8	64%

Based on the results presented one can draw the conclusion that the generally most time efficient solution is to execute the CFD simulations according to the Single scheme. E.g. executing only one process on each computer; leaving the second core virtually unused. For the lower amount of processes the Single setup require two times the computers that the Dual setup. This effectively doubles the hardware investments, but using two times the processes also doubles the software license costs. This issue was investigated in [22] [23] where the conclusion was that the cost of adding hardware is preferred to the cost of the added software licenses.

For Fluent and CFX the advantage seems to decrease as the number of processes increase but then raise again at 16+ processes, this statement is true for all but Fluent M2. As Openfoam goes the difference is even larger, here the Single setup outperformed the Dual for all number of processes with between 29% and 40% (on average 32%), although this does not include the Openfoam M3 Single 2 and Dual 4 simulations since they did not fit completely into memory.

6 Analysis and Discussion

The most notable results from the investigation are those presented in figure 5 and table 4 – In these cases the application and model combinations execute faster with one core inactive on each compute node, i.e. when one process (P) is executed on each one of the dual core computer nodes (N) compared to when two processes are executed on the same number of nodes.

The main issue is to determine if this behavior is dominated by hardware bottlenecks or the software scalability. If an underlying mathematical or programming intricacy causes behavior, the application should behave *the same as long as the number of processes are constant*. If all resource demands are satisfied the Dual(P) and Single(P) runs would execute in the same amount of time. Since this is not the case to try to identify the resource(s) that Dual(P) uses more of than Single (P). We now continue with a more detailed analysis looking the three candidate resources suggested in section 2 above.

6.1. Network Usage

The investigated applications at hand communicate on a process to process basis. All processes can communicate with all other processes as well as carry out collective MPI operations, such as broadcasts and scatter-collect operations. A process in a Single setup that sends a message to another process will always send it thru the switch (1). Where Switch represents the amount of traffic traversing the switch, N is the number of nodes, P is the number of processes and M is the average number of Megabytes transmitted by each process to each other process.

$$Switch_{Single} = (N^2-P) * M \quad \text{(eq. 1)}$$

A process in a Dual setup, however, will carry out a portion of its communication over the loopback interface but the larger part still goes through the switch.

$$Switch_{Dual} = (N^2-2P) * M \quad \text{(eq. 2)}$$

When comparing Eq. 1 and 2 it is obvious that the switch is subject to a higher utilization in the Single setup then in the Dual. However, in the Dual setup both processes on a computer node share the same network connection as well as the internal busses. The average load on each computer node is calculated as (switch/N)*(P/N). Nevertheless, consumed bandwidth is quite low. The Fluent M2 in 16 process Single configuration had the highest measured computer node communication 7.77 megabyte/s. It is one out of ten configurations that had a node communication of more than 5 megabyte/s i.e. 40 megabit/s. All CFX and Openfoam models had a per node communication less than 2.5 Megabyte/s.

Based on these numbers it is highly unlikely that the single process per machine performs faster than two processes per machine depends on the network resources. The increase in network utilization might impact the execution time in a negative way but not at all on the scale observed.

6.2. Memory Footprint

The investigated applications use a technique called bulk synchronous parallel [24], as do the majority of the industrial simulation codes in the CFD and FEM areas. This means that these applications at the global level hold a large matrix in memory of #E number of elements, where each element has a size of E_{size} in memory. The total memory size of the matrix is then $\#E * E_{size}$. Considering the inherent memory footprint M of the application, one could naively expect the memory usage per process to scale according to eq. 3 below, where P is the total number of processes.

$$M + (\#E * E_{size}/P) \quad \text{(eq. 3)}$$

However, all three applications use some optimization schemes to obtain higher performance. To avoid using the network too much some boundary data is stored with several processes. Instead of communicating, which in best case results in the introduction of network latencies and in the worst case bandwidth depletion, some elements are calculated and stored locally by several processes. The memory demands can thus more correctly be described by eq. 4, where Δ represents the amount of redundant information stored for optimization reasons.

$$M + (\#E * E_{size}/P+\Delta)$$

(eq. 4)

As can be seen in figure 9 the amount of memory does not decrease linearly with the number of processes. For many of the cases the per computer memory usage actually increases with the number of nodes. Evidently RAM-memory shortage cannot explain why the applications execute faster when only one process is placed on each dual core computer instead of two.

6.3.Memory Bandwidth

Despite that each process is identical in both execution and memory demands regardless of it being executed by itself on a compute node or together with an additional process, there is an important difference in technical context. Even if the memory allocated for two processes still easily can be fitted into the main memory (see above), the number of memory accesses will be (more than) doubled.

At the same time there is a major difference between the single and dual setup due to the shared L2 cache of the Intel Core2duo architecture. In the Single setup each process has a 4 megabyte L2 cache while in the Dual setup it has to be shared between the two processes, and the processes works on two different datasets. Thus, while the number of memory accesses *increases* by a factor of two or more (inter process communication among other things will create some extra load), the available L2 cache will in practice *decrease* by a factor of two. It is consequently reasonable to suspect that this change will give raise to additional delays not present in the single execution setup. When the STREAM benchmark was used to measure the sustained bandwidth it never exceeded 4080 megabyte/s, see section 3.1 for more details.

7 Conclusion

In this paper we investigated the impact multicore processor based clusters has on the execution of industrial simulations, namely computational fluent dynamics (CFD). Almost all new cluster installations are equipped with multicore processors, but according to our results it is far from trivial to determine if this is in fact a blessing or a curse.

The most significant finding was that five of the models, ran faster when only one process (P) were executed on each multicore node (N). When two processes were executed on each node the execution time was increased by between 6.5 and 64% with a median increase of 10% in these cases. In general the trend is that at a low number of nodes two processes per computer, i.e. one process per core, outperforms once process per node, but as the number of nodes and processes increase the gap closes and eventually the single placement strategy wins.

We evaluated four different possibilities, application scalability as well as processor, memory and inter process communication capacity as suggested by [6] as well as [7]. Application scalability was ruled out early since there were no scalability issues with the number of processes that were executed with one per computer node instead of two. The inter process communication capacity were also ruled out as a major contributor. Turning to the memory and memory bandwidth utilization it is obvious that the shared FSB and L2 cache of the Intel Core2duo architecture affects the performance. In the Single setup each process has a 4 megabyte L2 cache which in the Dual setup is shared and they work on two different datasets.

Ultimately we draw the conclusion that it is not obvious that utilizing more than one core of a dual core processor is beneficial. Furthermore we know that there are several cases where it is a great deal better to only use one core, especially when factoring in the per process license costs. The reason behind this (technical) behavior is still unknown although it is likely that it is linked to the memory hierarchy, and it is highly unlikely that it depends on the inter-process communication or the processor utilization.

8 References

[1] N. Singer, ”More chip cores can mean slower supercomputing,” *Sandia National Laboratories*, January 2009.

[2] J. Held, J. Bautista och S. Koehl, ”From a Few Cores to Many: A Tera-scale Computing Research Overview,” Intel white paper, 2006.

[3] L. A. Polka, H. Kalyanam, G. Hu och S. Krishnamoorthy, ”Package Technology to Address the Memory Bandwidth Challenge for Tera-scale Computing,” *Intel Technology Journal*, vol. 3, nr 11, 2007.

[4] L. Liu, Z. Li och A. H. Sameh, ”Analyzing memory access intensity in parallel programs on multicore,” i *International Conference on Supercomputing*, Island of Kos, Greece., 2008.

[5] S. K. Moore, ”Multicore is bad news for supercomputers,” *IEEE Spectrum*, November 2008.

[6] R. Buyya, High Performance Cluster Computing, New Jersey, USA: Prentice Hall, 1999.

[7] F. Vaughan, D. Grove och P. Coddington, ”Communication Performance Issues for Two Cluster Computers,” i *Proceedings of the Twenty-Sixth Australasian Computer Science Conference*, Adelaide, Australia, 2003.

[8] T. Sterling, D. Becker, D. Savarese och J. Dorband, ”Beowulf: A Parallel Workstation For Scientific Computation,” i *Proceedings of the 24th International Conference on Parallel Processing*, Oconomowoc, US, 1995.

[9] A. Boklund, C. Jiresjö och S. Mankefors, ”The Story Behind Midnight, a Part Time High Performance Cluster,” i *Proceedings of the 2003 international conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 2003.

[10] J. L. Hennessy och D. A. Patterson, Computer Architecture: A Quantitative Approach, 4 red., US: Morgan Kaufman Publishers, 2006.

[11] Intel, ”Intel Museum,” [Online]. Available: www.intel.com/museum/.

[12] L. Peng, J.-K. Peir, T. K. Prakash, Y.-K. Chen och D. Koppelman, ”Memory Performance and Scalability of Intel's and AMD's Dual-Core Processors: A Case Study,” i *Performance, Computing, and Communications Conference*, New Orleans, USA, 2007.

[13] J. D. McCalpin, ”Sustainable Memory Bandwidth in Current High Performance Computers,” 1995. [Online]. Available: http://www.cs.virginia.edu/~mccalpin/papers/bandwidth/bandwidth.html.

[14] C. Hristea, D. Lenoski och J. Keen, ”Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks,” i *ACM/IEEE conference on Supercomputing*, San Jose, USA, 1997.

[15] Ansys, ”CFX application,” [Online]. Available: http://www.medeso.se/software/ansyscfx.html.

[16] Ansys, ”Fluent application,” [Online]. Available: http://www.fluent.com/software/fluent.

[17] OpenCFD, ”Openfoam Application,” [Online]. Available: http://www.opencfd.co.uk/openfoam/.

[18] S. Perzon och L. Davidson, ”On CFD and transient flow in vehicle aerodynamics,” SAE Technical paper, 2000.

[19] L. T. Yang, X. Ma och F. Mueller, ”Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution,” i *Proceedings of the ACM/IEEE Super Computing Conference*, 2005.

[20] M. Frye, ”Debugging code with strace,” *RedHat Magazine,* August 2005.

[21] A. Boklund, C. Jiresjö, S. Mankefors-Christiernin, N. Namaki, L. Gustavsson-Christiernin och M. Ebbmar, ”Performance of Network Subsystems for Technical Simulation on Linux Clusters”, *Conference on Parallel and Distributed Computing and Systems*, Phoenix, USA, 2005.

[22] Boklund, N. Namaki, S. Mankefors-Christiernin, J. Gustafsson och M. Lingbrand, ”Dual Core Efficiency for Engineering Simulation Applications”, *Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 2008.

[23] A. de Blanche, S. Mankefors-Christiernin, ”Minimizing Total Cost and Maximizing Throughput - A Metric for Node versus Core Usage in Multi-Core Clusters,” i *International conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 2010.

[24] R. H. Bisseling, Parallel Scientific Computation: A Structured Approach using BSP and MPI, Oxford: Oxford University Press, 2004.

Combining Cache Aware Scheduling with Lazy Threads

Yosi Ben-Asher[1] **and Gil Kulish**[1]
[1]CS Department, University of Haifa, Haifa, Israel

Abstract— *We consider two factors that can dominate performances of fine grain parallel programming on multicore machines:*

- *Cache coherency protocols, which preserve cache coherency and by this, add large overhead.*
- *The number of real kernel threads that are used to execute the possibly large number of program threads explicitly generated by the parallel constructs of the program.*

As for the first factor we designed a cache aware scheduling scheme which, based on memory profile, schedules threads such that cache misses are minimized. As for the second factor, we implemented a lazy thread system, which replaces threads with loop iterations and function calls- minimizing the number of real threads spawned throughout the execution. These two techniques may be conflicting each other since by reducing the number of real threads that are generated we reduce the freedom degree of the cache aware scheduler to minimize cache misses. We consider ParC a programming language that is similar to OpenMP but supports a more generalized scoping rules than OpenMP, and designed a lazy thread system for it, enhanced with cache aware scheduling. Our results prove that cache aware scheduling can be effective even with very aggressive lazy thread optimizations. The implementation of the scheduling system is optimized for the MESI cache coherency protocol.

Keywords: Multicore, Cache, Lazy threads

1. Introduction

We consider parallel programming over a multicore machine wherein shared memory is simulated by maintaining a cache coherency protocol, such as MESI [14]. For parallel programming, we consider two parallel constructs that can be freely nested: $parfor(i = 0; i < N; i++)\{body\}$- A parallel version of C for-statement generating a separate thread for each iteration i. $parblock\{body_1\} : \ldots : \{body_k\}$- A parallel version of C block-statement generating a separate thread for each $body_i$ The threads generated by the execution of such programs should be executed in parallel by different cores of the machine so that parallelism is obtained. Accessing variables by such a code might imply use of shared memory since the same variable can be accessed by different threads executed on different cores.

One factor that can easily reduce the speedups of parallel programs executed on a multicore machine is the overhead involved with simulating the shared memory. Typically, shared memory over a set of cores is simulated via a cache coherency protocol, e.g., MESI [14]. MESI is a distributed protocol that ensures values of shared variables in the different cores' caches and in the main memory are consistent. In particular, MESI ensures that all the different copies of a shared variable stored in the different caches and the main memory have the same value when accessed. This is done by invalidating copies at remote caches every time a core updates a shared variable in its cache and updating the memory. Consequently, a cache miss (accessing an invalid copy of a shared variable) results in a bus transaction needed to fetch a value from the main memory. A cache line is always in one of the following states: Invalid, Shared (other caches contain a copy), Exclusive (only this core contains a copy), and Modified (local copy differs from the copy in the main memory). Consequently read/write operations on shared variables can lead to a bus-transaction that may slow down the computation significantly (we measured a factor of 40 times slower for accesses that are not cache-hit). Thus it is important to reduce the number of shared memory references that are cache misses and incur bus transactions to the main memory. Note that accessing two shared variables separately by two threads that are executed in different cores typically results in cache misses and bus transactions

In this respect it is natural to consider the possibility of scheduling the threads that are generated during the execution of a parallel program, such that the overhead of accessing shared variables is minimized. Consider for example the following parfor executed on a multicore machine with $p = 2$ cores:

```
parfor(int i=0;i<=n;i++)
  if(i < n/2)
     for(int j=0;j<i;j++) A[j]+=f(j);
  else for(int j=0;j<i;j++) B[j]+=g(j);
```

Let T_i be the thread generated by the i'th iteration of the above parfor and consider two possible schedules (where $T_i \mid\mid T_j$ indicates that T_i executed by the first core and T_j by the second core:

$$\begin{array}{lcl} T_0||T_1; & & T_0||T_{n/2}; \\ T_2||T_3; & & T_1||T_{n/2+1}; \\ T_4||T_5; & and & T_2||T_{n/2+2}; \\ \ldots\ldots & & \ldots\ldots \\ T_{n-1}||T_n; & & T_{n/2-1}||T_n; \end{array}$$

Clearly, the leftmost scheduling will result in many cache

misses and MESI bus transactions as the first half of threads are executed in parallel repeatedly updating the shared array $A[]$ and similarly for the second half of the threads all accessing $B[]$. The rightmost scheduling is significantly better since when two threads are executed in parallel then most of their shared memory references point to different arrays. In this work, we consider ways to compute this $cache\ aware\ scheduling$ and execute it for a parallel program. We propose a specific technique that is based on memory profile analysis combined with simulation of the MESI protocol.

Our cache aware scheduling is part of the thread system that executes the parfor iterations and the parblock bodies of the parallel program since it affects the scheduling decisions made by the underlying thread system. Thus, cache aware scheduling should be evaluated in the context of the underlying thread system and not as a separate optimization. For example, scheduling strategies in thread systems are used to balance the execution time of the threads between the different cores. Thus, cache aware scheduling may potentially affect the ability of the underlying thread system to obtain good load balancing between the core. In this work we consider another aspect of thread systems called "Lazy Threading", which can potentially conflict with the ability to effectively execute cache aware scheduling. Basically, lazy threading maximizes the number of program threads (parfor iterations and parblock bodies) that are executed sequentially as loop iterations (for parfor iterations) or as local function calls (for parblock bodies), as opposed to kernel threads. This is an important feature of thread systems proposed by Goldstein et al. [10] implying that the possibly large set of program threads can be statically packed into a smaller set of threads which are actually executed by the underlying thread system. A common simple case of lazy threading is known as "chucking" where the n iterations of a $parfor(i = 0; i < n; i++)\{\ldots\}$ are executed in parallel using p real threads each executing a chunk of $\frac{n}{p}$ iterations sequentially. For example, the thread system of OpenMP automatically chunks all the iterations of every parfor ignoring possible dependencies that might exists between the different iterations of this parfor (though OpenMP allows the user to specify the number of threads a given parfor should use). In this way the large overhead involved with thread creation, termination and preemption is reduced. However, lazy threading can potentially reduce the effectiveness of cache aware scheduling since it eliminates most of the program threads reducing the number of possible scheduling available for the cache aware scheduling. Figure 1 illustrates this problem showing a program that spawns eight threads $T3 \ldots T10$ via two parfor statements to be executed on a multicore with $p = 2$ cores. Applying lazy threading to the program of figure 1 will result in clustering the threads $T3 \ldots T6$ and $T7 \ldots T10$ as two real threads, which, as depicted in figure 1, will prevent efficient cache aware scheduling. As depicted in figure 1 right bottom side, the scheduler can potentially schedule $T3 \ldots T10$ such that no cache misses occurs if it avoides lazy threading. Thus, in this work we would like to see if the reduced set of threads that remains after lazy threading and even a highly aggressive form of lazy threading still allow efficient cache aware scheduling.

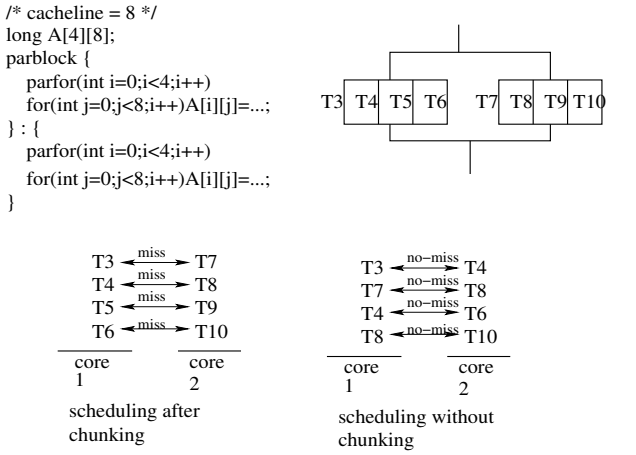


Fig. 1: Lazy threads prevent cache aware scheduling

We have implemented the cache aware scheduling and the aggressive lazy threading for ParC [3]- a parallel programming variant of C/C++ which is similar to OpenMP commonly used for programming multicore machines. Thus, the technique proposed here is general and can be used for OpenMP and other thread-systems running parallel programs over multicore machines.

The contributions of this work are as follows:

1) Developing a cache aware scheduling scheme for MESI protocol, which to the best of our knowledge, has not been proposed elsewhere.
2) Creating an aggressive lazy threading scheme that maximizes the number of program threads that are executed sequentially as function calls and loop iterations. Previously suggested schemes for lazy threading did not handle parfor constructs and while-loops whose termination depends on shared variables modified by other threads.
3) Showing that the combination of lazy threads and cache aware scheduling is beneficial.

2. Thread systems and lazy execution of threads

In order to execute a parallel program by a multicore machine it is necessary to use a thread system that can execute the multiple program threads generated by the parallel program on the small fixed number of available cores (typically 4-8). Basically such a thread system consists of two

queues of threads: ready-queue containing thread's control blocks that can be executed and a suspend-queue containing control blocks of threads that are suspended waiting for some event to occur. Threads are pulled from the ready-queue, executed for a while, stopped, and then their execution state is stored in the ready-queue again. Threads that spawn new threads due to nesting of parallel constructs are placed in the suspend-queue until all its descendants threads terminate. This general notion of ready-queues is basically valid for any thread system. We use the term $scheduler$ to describe the module that is responsible for selecting the next set of threads from the ready queue and assigns the thread to an available core. Clearly, for a given state of the ready-queue there can be many possible scheduling strategies: Round-robin- selecting the oldest thread in the ready queue and assigning it to the next available core. Random- selecting k threads at random where k is the current number of free cores. Random selection can improve load balancing between the cores as described in [2]. Multiple-queues-maintaining p (number of cores) ready/suspend-queues one for each core and exporting newly spawned threads between the different queues. In this case the scheduling strategy includes rules for selecting the next thread for each core from its ready-queue. It also includes rules for exporting newly spawned threads between the different cores. For example, we may decide to partition new threads spawned by a $parfor$ evenly between the cores' ready-queues or send them to a core whose ready-queue contains the least number of threads out of all cores. There are several alternatives to switching between a running thread and a suspended thread waiting inside the ready-queue. One option is to use time interrupts and switch threads after some fixed time quantum, another alternative is that the compiler will insert explicit instructions in the code of the threads causing it to explicitly switch to another thread. Thus, a thread system (TS) is a software module that consists of: ready-queue, suspend-queue, scheduler, context_switch_mechanism and a source of new threads which is the execution of the parallel program. A multiple queue thread system is a fixed set of TS_i and rules for exporting new threads to the different ST_is.

In this work we, use a three level TS model containing the following levels: Kernel threads generated using Pthreads [12] with one thread per core, User threads- generated by Pth [8] forming a lighter thread levels driven by explicit Pth context-switch instructions. A User-Level TS is generated once per each Kernel Thread. Program threads- that are even lighter threads that run to completion in lazy thread mode that is explained next. A single thread at each user level TS will execute the lazy TS. Lazy execution of threads and Parallel loops load balancing are known concepts and basically imply that a maximal number of parfor iterations will be executed sequentially without generating any thread.

Feitelson and Rudolph addressed the issue of how a multiprocessor should divide its processing resources among competing jobs. [9]. Since then many researches were done in the field of clustering threads into gangs and scheduling them to run over multiprocessors. Recent work of Nikolopoulos and Polychronopoulos [13] addressed the issue of of data locality. Calandrino and Anderson [6] [5] attempted to prevent a case in which two threads that use different areas of the memory and use the cache intensively run concurrently. Thibault, Namyst and Wacrenier [15] [16] address the problem of poor scheduling APIs. Chu, Ravindran and Mahlke [7] proposed a profile-guided method for partitioning memory accesses across distributed data caches. "Lazy threads" were proposed in [10] but in a limitted form using fork operations only. OpenMP suggests several keywords that allow the programmer to predefine the scheduling scheme [4]. several works done in the field such as [1] suggest to preserve spatial locality, by assigning contiguous chunks of iterations to the same thread whenever possible.

3. Proposed lazy thread mechanism

The simplest form of lazy threads it to partition the n iterations of a $parfor(i=0;i<n;i++)S_i$ to p chunks of n/p iterations that are executed sequentially:

$$parfor(k=0;k<n;k+=n/p)$$
$$\quad for(i=k;i<k+n/p;i++)\ S_i;$$

We use a function $selector(f,t,S_i)\{\ for(i=f;i<t;i++)\ S_i;\}$ to execute a chunk of parfor iterations $i=f\ldots t-1$. There are two cases in which a chunk cannot be executed sequentially using loop iterations. In the first, one of the $parfor$ threads (say i) of a chunk (from f to t) executes a context switch instruction. Note that the ParC compiler should insert a context switch instruction only inside those while-loops whose condition may depend on the value of shared variables. In this case, it might happen that such a while-loop (of ParC thread i) depends on a "releasing" assignment that will be executed by one of the remaining iterations (threads) of this chunk $i+1\ldots t-1$. Thus, in case of a context switch, the selector function might be forced to send the remaining $t-i-1$ iterations to be executed concurrently on a new user thread, in order to allow the execution of the aforementioned "releasing" assignment. The second case occurs when the i'th iteration spawns a new parallel construct. Here, a new selector function should be called by the current selector. The problem in this case is that a local descendant of this new parallel construct might be dependent on one of the remaining $i+1\ldots t-1$ siblings of the spawning thread i. However, the remaining $i+1\ldots t-1$ siblings will not be executed until the new call returns.

In order to reduce the number of user threads that are used during the execution, we must take care not to create a new kernel thread whenever a spawn occurs. A spawn request in thread i can be executed in one of two ways:

either by sending the remaining $i+1 \ldots t-1$ threads to be executed by another user thread, or by first executing the new spawn locally (sequentially). Then, when all its descendants terminate, local execution of the remaining $i+1 \ldots t-1$ threads can be carried out sequentially. In our scheme, we propose to delay the decision about how the spawn should be executed, first we try to perform the remaining threads sequentially, without spawning new threads, until we discover that a descendant of the current thread i has performed a context switch. If all the spawned descendants of i have terminated without executing a context switch, then the execution of the remaining threads $i+1 \ldots t-1$ can continue as if the spawn in i did not occurred. Otherwise, a new thread will execute the remaining threads must be created. This is because of possible dependency (through shared variables in a while-loop) of this thread on one of the remaining threads that can only be resolved by executing a context switch.

The usefulness of this scheme for lazy threads is demonstrated in the following example. Consider the execution of the following program using the proposed scheme. There is a spawn in the first thread $i == 0$. Thus, the range $1 \ldots \frac{n}{p} - 1$ is stored in the remaining stack. The chunks of the inner $parfor$ contain a while-loop $while(x)$, causing a context switch to occur. Thus, the remaining iterations $1 \ldots \frac{n}{p} - 1$ will be executed concurrently by another thread. As a result, thread $i == 1$ will set $x = 0$ and free the threads of the inner $parfor$ that are waiting in $while(x)$. Thread $i == 0$ will execute $y = 0$, allowing the termination of all the $while(y)$ in threads $i = 2 \ldots n$. This is close to the optimal scheduling, as most of the while-loops' flags will be reset as soon as thread $i == 1$ executes $x = 0$.

```
int x=1,y=1;
parfor(int i=0;i<n;i++) {
  if(i == 0){
    parfor(int j=0;j<n;j++){ while(x);}
    y = 0;
  }
  else if(i== 1) x = 0;
  else while(y);
}
```

4. The cache aware scheduling algorithm

In here we outline the main stages and detailes involved with the cache aware scheduling. The cache aware scheduling is executed using the following steps: Program execution step- the program is compiled using the ParC compiler that generate C++ program with suitable function calls to spawn threads. The compiler implements the aggressive lazy thread technique described earlier. The resulting code is linked with the thread system and is executed in parallel on every core. Figure 2 illustrates a snapshot in the execution of a program where the lazy threading generated four user-threads $T_1, T2, T3, T4$ such that T_1, T_3 reside in core-0's ready queue and T_2, T_4 reside in core-1's ready-queue. Note that this implies that any profile information collected is relevant only to the specific scheduling that occurs during the program's execution.

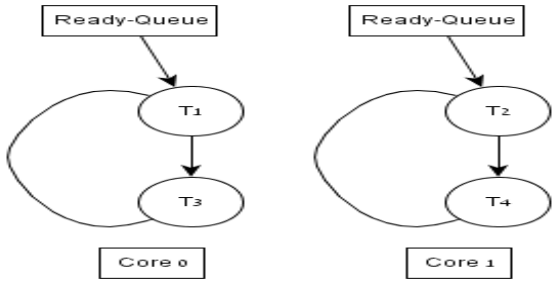


Fig. 2: Executing a program

Cache profile step- where each memory reference is instrumented such that upon execution, a logfile containing cache access statistics is generated. For every cache line L_j and user thread T_i, we record the number of times T_i updated a variable in L_j.(as depicted in figure 3). This number is called $T_{i,j}$. In order to capture potential cost for threads that run from different cores, we assume that each thread is executed from a different core. The cache simulation used here does not include cache associativity. Time intervals of cache references are used to determine when two threads have a possible cache conflict. Threads that originate from the same parallel constract, e.g. , chunks originating from the same parfor, are also considered conflicting. The potential conflict cost generated from two conflicting threads T_{i1} and T_{i2} is the sum of all minimums between($T_{i1,j}$ and $T_{i2,j}$) where j represent conflicting cache lines for threads T_{i1} and T_{i2}.

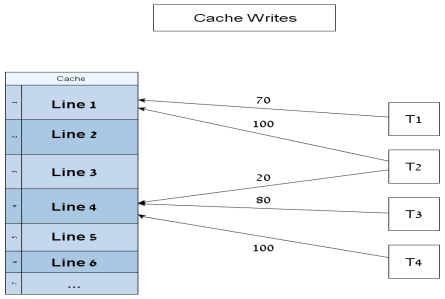


Fig. 3: Cache statistics obtained for the execution of figure 2.

MESI profile step- where each memory reference is instrumented such that upon execution, a logfile recording MESI's bus transactions will be generated. The simulation refers

to different threads as different cores. During execution the MESI operation in each thread is fully simulated and bus transactions resulting by accessing shared variables are recorded. The information is collected such that for every two threads $< T_i, T_j >$ we record the number of bus-transactions resulting from a consecutive accesses of T_i and T_j to the same cache line. For example assume that T_i updates a shared variable x residing in cache line L_8 invalidating L_8 in the remote core where T_j is executed. Next T_j attempts to read y which is also mapped to L_8. Thus, T_j will cause a MESI bus-transaction which will increment the counter of bus-transactions between T_i and T_j. These type of threads will be defined as conflicting threads. The cache simulation used here does include cache associativity.

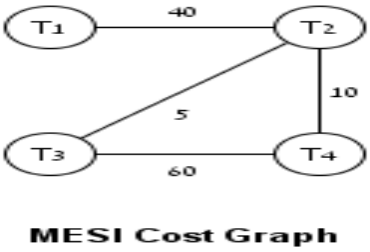


Fig. 4: MESI statistics obtained for the execution of figure 2.

Unified profile step- where the Cache profile information and the MESI profile information are unified. This is done by first summing the common cache accesses between every two conflicting threads $< T_i, T_j >$ based on the information in figure 3. Next we factor in conflicting threads information collected during the MESI profile, for every two conflicting threads $< T_i, T_j >$ we add the half of the MESI statistics as depicted in figure 5. Note that both profiles are needed

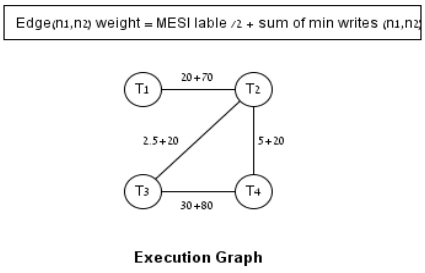


Fig. 5: Unifying MESI profile and the cache profile for the execution example of figure 2. $edge(T_{in}, T_{im}) = 0.5 * MESI(T_{in}, T_{im}) + \sum_{j=1}^{cache\ lines} minimum\ cache - profile(T_{in,j}, T_{im,j})$

since:

- The MESI profile is more accurate than the cache profile since it counts real bus-transactions skipping cache references that do not lead to bus transactions. For example, consecutive updates to a variable that is in state Exclusive should not be counted.
- The MESI profile can be misleading as well since it can only count bus transactions that occur between threads that run concurrently. However, since the actual cache aware schedule that will occur during the final run of the program might be different than the original one used for collecting the MESI profile, we might ignore cache misses between threads that were not executed is parallel in the original schedule.
- The reason for dividing the MESI profile values by two when factoring it with the weights of the cache profile is due to the fact that the Cache profile represents a set of possibly bad events that can occur under any schedule while the MESI profile is relevant only for a specific schedule.

The unified profile also includes the number of memory references executed by each thread (associated with every node) Graph partition step- where we use a graph partition package [11] to partition the nodes of the unified profile graph into p clusters of threads such that:

- The total weight of edges connecting threads between different clusters is minimized.
- The total weight of nodes in each cluster is about the same.

In the cache aware schedule all the threads that were allocated to the same cluster will be schedule to the same core. The graph partition thus obtains good load balancing while minimizing the number of bus transactions that are likely to occur between threads that are executed on different cores. Figure 6 depicts the resulting graph partition of the unified profile graph.

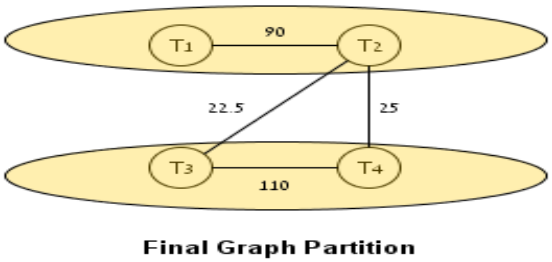


Fig. 6: Graph partition of the unified profile graph of figure 5.

Final execution step- where the resulting graph partition is recoreded into a configuration file that is used by a future run of the program with possibly a different input. The program must be recompiled with a suitable flag to avoid the

overhead of the instrumented code used to collect the profile information. Figure 7 depicts the resulting scheduling that will occur when the program will be executed with specific directions for the scheduler reflecting the graph partition obtained by the profile gathering stage. It follows that the resulting cache aware schedule reduces the expected number of bus-transactions from $90 + 110 + 22.5$ to $22.5 + 25$. Obviously, this is only an estimation and does not reflect real numbers of bus transactions that will occur in an actual execution.

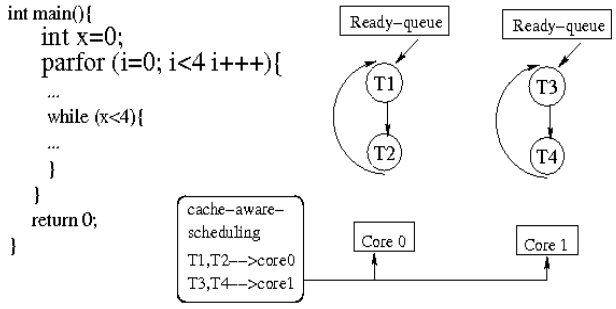


Fig. 7: New execution guided by the graph partition of figure 6.

A final aspect related to the proposed technique is how threads are labeled during the profile gathering stage and in during actual execution of the program that is using the resulting cache aware scheduling. Technically the labels of threads should allow us to:

- Uniquely identify threads resulting from the different parfors and parblocks of the program.
- Match between the set of threads generated during the profile-state and the set of threads generated by a "final-run". Note that since the final-run will use a different input than the one used for the profile run then the set of threads that it generated can be different than the one for which the cache aware scheduling was computed with,

We thus use thread-labels that are based on the static nesting structure of the program which is the same for both runs as follows:

- Each parfor/parblock in the source code is assigned a unique label, e.g., PF1,PB2, and so forth.
- Upon execution when a thread is generated its label is concatenated to the label of the thread that spawned it, e.g., for a three level nested parfor the innermost thread labels will be of the form PF5.PF10.PF17.
- Iterations numbers are also dynamically concatenated to the labels according to the range of iterations of the $selector()$ functions, e.g., $PF1_10 : 20.PF5_77 : 198$.
- Sequential executions of a parallel construct, such as parfor will result in a rising sequential iteration number.

If a configuration file with a cache aware scheduling is available, the scheduler at the current run will use the labels to match threads to cores and implement the scheduling in the order specified in configuration file. Threads whose labels do not match the labels in the configuration file are scheduled in the core that scheduled one of their ancestors according to their label.

5. Experiments

We describe and analyze the results of running 7 known benchmarks to compare OpenMP with ParC. The benchmarks were downloaded for OpenMP and implemented for ParC. We were extra careful to ensure that the ParC code will be true to the original OpenMP source code, meaning, no changes were made to the structure of the code in order to give synthetic advantage to ParC. We used 8 kernel threads for both OpenMP and ParC assuming two hardware threads per core. The experiments in this section were conducted on Core. i7 64 bit machine with Core. i7 920 processor, 48 Giga bytes of memory and four cores. In the following experiment, we measure the performance improvement gained from using the cache aware scheduling (lazy scheduling was turned off by using eparfor). Not optimized, the code bellow will spawn multiple threads, which most probably access the same cache line in the same time from different cores. After running the profiler, the threads will be grouped in such a way that threads which share a cache line will be executed from the same core.

The following experiments are public benchmarks used to compare OpenMP vs ParC performance. ParC showed a consistent advantage of X 1.2 shorter execution time on various matrices sizes. For example the following table contains the results for matrix multiplication:

Matrixes sizes	2000	2500	3000
OpenMP	24	49.5	90
ParC	20	41.5	75
Improvement Ratio	1.2	1.19	1.2

Table 1: OpenMP VS ParC Matrix multiplication comparison.

Other benchmarks include: NASA's NPB2.3, Molecular Dynamics (MD), a Clustering Algorithm, Game Of Life (GOL), Hopfield Neural Network (HNN), and Discrete Cosine Transform (DCT), Table 2 shows the improvement obtained due to the use of our cache aware scheduling summery of all of the above benchmarks:

References

[1] Eduard Ayguade, Bob Blainey, Alejandro Duran, Jesús Labarta, Francisco Martínez, Xavier Martorell, and Raul Silvera. Is the schedule clause really necessary in openmp? In *Proceedings of the International Workshop on OpenMP Applications and Tools 2003, volume 2716 of Lecture Notes in Computer Science*, pages 69–83, 2003.

Benchmark	#Lines - OpenMP	# Lines - ParC	OpenMP Time in Seconds	ParC Time in Seconds	Improvement Ratio
Mat Mul	69	69	90	75	1.20
NPB 2.3	926	1008	130	24	5.42
MD	626	636	96	33	2.91
Cluster	294	320	354	635	0.56
GOL	270	260	6.1	1.1	5.55
Hopfield	255	235	54	18	3.00
DCT	240	240	52	17	3.06

Table 2: OpenMP VS ParC Comparison.

[2] Y. Ben-Asher., A. Cohen, A. Schuster, and J.F. Sibeyn. The impact of task-length parameters on the performance of the random load-balancing algorithm. Technical report, Proc. 6th International Parallel Processing Symposium, 1992.

[3] Y. Ben-Asher, D. G. Feitelson, and L. Rudolph. Parc: An extension of c for shared memory parallel processing. *Software practice & Experience*, 26(5):581–612, 1996.

[4] Addison C., LaGrone J., Huang L., and Chapman B. Openmp 3.0 tasking implementation in openuh. Open64 Workshop at CGO 2009, 2009.

[5] John M. Cal and James H. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study ?, 2008.

[6] John M. Cal, James H. Anderson, and Dan P. Baumberger. A hybrid realtime scheduling approach for large-scale multicore platforms. Univ. of North Carolina at Chapel Hill.

[7] Michael Chu, Rajiv Ravindran, and Scott Mahlke. Data access partitioning for fine-grain parallelism on multicore architectures. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 369–380, 2007.

[8] RALF S. ENGELSCHALL. Gnu portable threads (pth), 1999. http://www.gnu.org/software/pth/, ftp://ftp.gnu-/org/gn.

[9] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306 – 318, 1992.

[10] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of parallel and distributed computing*, 37:5–20, 1996.

[11] George Karypis and Vipin Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, The University of Minnesota, 1995.

[12] Frank Mueller. A library implementation of posix threads under unix. In *In Proceedings of the USENIX Conference*, pages 29–41, 1993.

[13] Dimitrios S. Nikolopoulos, Eleftherios D. Polychronopoulos, and Theodore S. Papatheodorou. Efficient runtime thread management for the nano-threads programming model. In *Proc. of the Second IEEE IPPS/SPDP Workshop on Runtime Systems for Parallel Programming, LNCS*, 1998.

[14] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12:348–354, January 1984.

[15] Samuel Thibault. A flexible thread scheduler for hierarchical multiprocessor machines. *CoRR*, abs/cs/0506097, 2005.

[16] Samuel Thibault, Raymond Namyst, and Pierre andr? Wacrenier. Building portable thread schedulers for hierarchical multiprocessors: the bubblesched framework. *ACM*, 8:00154506, 2007.

Tuning G-Ensemble to Improve Forecast Skill in Numerical Weather Prediction Models

Hisham Ihshaish*, Ana Cortés, and Miquel A. Senar
Computer Architecture and Operating Systems Department,
Universitat Autònoma de Barcelona, Barcelona, Spain
hisham@caos.uab.es, ana.cortes@uab.es, miquelangel.senar@uab.es

Abstract— *The process of weather forecasting produced by numerical weather prediction (NWP) models is complex and not always accurate. Moreover, it is generally defined by its very nature as a process that has to deal with uncertainties. In previous works, a new weather prediction scheme, Genetic Ensemble (G-Ensemble), was presented, which uses evolutionary computing methods. Particularly, it uses Genetic Algorithms (GA) in order to find the most timely 'optimal' values of model closure parameters that appear in physical parametrization schemes, which are coupled with NWP models. The presented scheme showed significant improvement of weather prediction quality and, moreover, the waiting time for an enhanced weather prediction result was reduced by executing a parallel G-Ensemble scheme over HPC platforms. In this work, however, we test the same scheme with different GA configurations regarding its Crossover type and ratio, and by variating its initial population size in order to get better predictions. The main concern behind this work is to provide a more detailed study on how the GA used in G-Ensemble scheme could be tuned depending on the available computational resources in operational scenarios. Finally, experimental results are discussed of a weather prediction case using historical data of a well known weather catastrophe: Hurricane Katrina that occurred in 2005 in the Gulf of Mexico. Obtained results provide significant enhancement in weather prediction.*

Keywords: numerical weather prediction; HPC; genetic algorithm; ensemble prediction; parameter estimation.

1. Introduction

It is generally agreed that weather has a widespread impact on people's personal and social lives, including their jobs, their recreation, their safety, and their property. When the weather is bad, many activities become more difficult to perform. Commercial transportation slows down on the roads, on the waterways, and in the air. Businesses of all kinds are interrupted by bad weather. Power plants and energy traders rely on knowledge of the weather to operate their equipment and to deliver power to consumers, government and business. Furthermore, accurate predicted weather variables are critically needed for other environmental modeling systems. For instance, wind direction and velocity variables are needed as precise as possible to predict the expansion direction and velocity of a fire propagation disaster predicted by wildfire models.

Weather forecasting that predicts future weather state evolution is realized mainly by Numerical Weather Prediction (NWP) models that are commonly solved by means of computing facilities. That is, a numerical weather prediction is the process of guessing the future state of the atmosphere based on current weather conditions. Mathematical models are used to do the job, which treat the atmosphere as a fluid. As such, the idea of numerical weather prediction is to sample the state of the fluid at a given time and use the equations of fluid dynamics to estimate the state of the fluid at some time in the future.

On the other hand, from a computational point of view, NWP models are considered as soft-real time large scale applications. The importance of having a certain degree of accuracy in the prediction in a certain time is a real challenge. Many factors may determine the accuracy of the predicted weather variables: the available computing power for model execution, the model itself, and the input data. Thus, ongoing research concentrates on methods to enhance the process of prediction and get results of this process faster.

However, and as most simulation software works with well-founded and widely accepted models, the need for input parameter optimization to improve model output is a well-known and often-tackled problem. Particularly, in environments where correct and timely input parameters cannot be provided, efficient computational parameter estimation and optimization strategies are required to minimize the deviation between the predicted scenario and the real phenomenon behaviour.

With the continuously increasing availability of computing power, evolutionary and parallel optimization methods, especially Genetic Algorithms (GA), have become more popular and practicable to solve the parameter problem of environmental models.

In [1], a study discussing the sensitivity of forecast skill to a set of NWP model closure parameters (input parameters)

This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974.

*Corresponding author.

†This paper is addressed to the PDPTA conference.

is provided. Furthermore, *G-Ensemble* prediction scheme is presented, which uses a GA to estimate 'optimal' values for these parameters for a certain forecast, in order to enhance forecast skill. The proposed scheme showed significant enhancement in prediction quality. In this work, more prediction results are presented and discussed regarding different configurations and scenarios of the used GA in the *G-Ensemble* prediction scheme. The aim of the presented work is to show how better predictions could be achieved by tuning the implemented GA in the *G-Ensemble* approach.

The rest of the paper is organized as follows: Section 2 gives an overview of NWP models, a NWP general scheme, and a brief description of the Weather Research and Forecasting Model (WRF), which constitutes the most commonly used model for weather and meteorological predictions. Section 3 discusses the predictability sources of error in NWP models and also describes the most widely used methods for NWP enhancement in practice. In section 4, *G-Ensemble*) is described briefly. Section 5 discusses experimental results obtained with a test case, where we compare our proposal with other enhancement methods. Finally, conclusions and future work are described in section 6.

2. Numerical Weather Prediction Models

Weather stems from the constant evolution of the atmosphere governed by physical laws. Using high-speed computers to solve a complex set of mathematical equations that represents the governing laws, NWP is a technique for simulating the atmospheric evolution in order to delineate the resultant weather changes. The variables involved in the equations include wind, temperature, pressure and moisture content. In principle, given the initial and boundary conditions, the atmospheric variables can be numerically solved as functions of time and form the basis of weather forecast. That is, NWP is described generally as "an initial-boundary value problem": given an estimate of the present state of the atmosphere (initial conditions), and appropriate surface and lateral boundary conditions, the model simulates (forecasts) the atmospheric evolution. The more accurate the estimate of the initial conditions, the better the quality of the forecasts.

Certain areas where atmospherical future conditions are to be predicted are represented by three-dimensional uniform-gridded-rectangles referred as domains or grids. The input data, which describe an estimation of the actual state of the atmosphere, are called initial conditions. Those initial conditions are assigned to all points of the grid. The horizontal distance between grid points is referred as the spatial resolution of both the initial conditions and prediction results. Regional models (also known as limited-area models, or LAMs) allow for the use of finer grid spacing (higher resolution) than global models because the available computational resources are focused on a specific area instead of being spread over the globe. This allows regional models to resolve explicitly smaller-scale meteorological phenomena that can not be represented on the coarser grid of a global model. Hence, a NWP model will predict the new values of the initial conditions over future time scale.

The first step of a NWP process is to extract initial conditions that are usually obtained from a global forecasting. These initial conditions are assigned to the domain grid points and, by means of the NWP model applied over a time line, at each pre-defined time period, a new 3-dimensional domain is produced having new (predicted) values of meteorological variables at all grid points.

The Weather Research and Forecasting model (WRF) [2] is a widely-used numerical weather prediction system, which is considered as a next-generation mesoscale numerical weather prediction model designed to serve both operational forecasting and atmospheric research needs.

WRF is composed of a variety of programs to facilitate the prediction process. It includes modules for global terrain data extraction, modules for real observation injection while model integration, and modules for output post-processing. It should be mentioned that although we have applied our methodology to WRF, the proposed strategy is a model-independent design, which could also be used with other existing NWP models such as the PSU/NCAR Mesoscale Model [3] known as (MM5).

3. Related Work

NWP models as well as the atmosphere itself can be viewed as nonlinear dynamical systems in which the evolution depends sensitively on the initial conditions. Moreover, weather prediction is, by its very nature, a process that has to deal with uncertainties. The initial conditions of a NWP model can be estimated only within a certain accuracy. During a forecast, some of these initial errors can amplify and result in significant forecast errors. Besides initial-condition error, weather and climate prediction models are also sensitive to errors associated with the model itself. In particular, the uncertainty due to the parameterizations of sub-grid-scale physical processes is known to play a crucial role in prediction quality (e.g., [4]). Prediction errors caused by the uncertainty in physical parameterizations is commonly referred to as model errors. Weather predictability errors are normally subject to two kinds of errors: initial condition errors and model errors.

As it has been stated before, in the case of initial conditions, input data is extracted from global forecasts. Normally, global forecasts are conducted using domains of lower grid resolutions (the distance between grid points is large). This is due to the computational power needed if the whole globe is to be predicted using finer grid spacing. As a result, interpolations are needed to extract initial conditions from lower resolution domains to assign them to local domains of

higher resolution. Unfortunately, this process is not perfect and the assigned values do not reflect the actual real state of the atmosphere. This problem is generally referred as the uncertainty of weather initial state.

On the other hand, physical parametrization is the representation of sub-grid scale physical processes, that is, some meteorological processes are too small-scale to be explicitly included in NWP models. Hence, parametrization enables the representation of these processes by relating them to variables on the scales (the points of the gridded domain) that the model resolves. For example, an important meteorological process is the surface flux of energy transmitted by the terrain which helps in enhancing the prediction of other important variables like near-surface temperature, sea surface temperature and even near-surface wind velocity variables. This process normally occurs in scales smaller than 1 kilometer, while NWP models predicts normally on domains of grid-scales higher than 1 kilometer. Parametrization is needed in such cases to represent this process on a certain domain scale.

By figuring out the main sources of error in predictability of NWP models, and over the past 20 years or so, stochastic or "ensemble" forecasting [5] became as a practical and successful way of addressing the predictability problem associated with the uncertainty in initial conditions. Moreover, several weather prediction centers have addressed this problem by developing operational ensemble prediction systems (EPS) (e.g., [6]). The main idea behind an EPS comes from the fact that the initial state of a certain variable should be seen as a probability distribution and not as a unique value, and thus, the ultimate goal of ensemble forecasting is to predict quantitatively the probability density of the state of the atmosphere at a future time. This is done by running multiple forecasts, each of which is initiated with small perturbations in the estimated initial conditions. Then, an ensemble forecast is usually evaluated in terms of an average of the individual forecasts (ensemble members) concerning one forecast variable, as well as the degree of agreement between various forecasts within the ensemble system, as represented by their overall spread [7].

However, and although it has been realized that there is a stochastic nature of physical parameterizations in ensemble prediction (predictability is sensitive to variations in physical parameters), it has not been straightforward to develop theoretically sound, and also practical, formulations for how to insert parameterization uncertainty into ensemble development [8], [9].

4. G-Ensemble

In this section, Genetic Ensemble (*G-Ensemble*) approach [1] for prediction enhancement is briefly described, as well as the set of the model closure parameters targeted for better estimation. The main objective of the presented scheme is to enhance prediction quality by improving the estimation of a set of NWP model closure parameters. The study is focussed on finding 'optimal' values of *Landuse* and *Soil* closure parameter (the land surface parameters and the impact they have are described in [10]). The optimization of these parameters will serve as a prove of concept of our method, which could be applied to other parameters. These parameters are found in land surface physical schemes (LSM) (e.g., [11], [12]) that are coupled to most NWP models. The proposed scheme consists of two phases: Calibration Phase and Prediction Phase (depicted in Fig.(1)).

Considering that t_i is the instant time from which the meteorological variables are going to be predicted, i.e. prediction is done within the period (t_i-t_{i+n}), Calibration Phase starts at a time prior to prediction time and ends at time 00:00 (t_i) of prediction period, i.e. calibration is done within the period (t_0-t_i). The process of closure parameter estimation in Calibration Phase proceeds as follows:

1) at the beginning of Calibration Phase (time t_0 in Fig. (1): a sample of the targeted parameter values from ensemble proposal distribution is generated (perturbations in closure parameter values);
2) the generated parameter values are inserted to the ensemble prediction model;
3) an ensemble of forecasts (the prediction model is different for each ensemble member regarding the targeted parameter values) is conducted to predict meteorological variables at time t_i, where real observations are available;
4) evaluation of a fitness function for each ensemble member is done at time t_i;
5) genetic algorithm functions (selection, crossover and mutation) are used to generate a new ensemble distribution from the set of combinations of closure parameters which score better predicting at time t_i;
6) the process is repeated iteratively until a predefined number of iterations, or an acceptable error value is achieved.

The used fitness function depends on the number of meteorological variables to be better predicted.That is, if the *G-Ensemble* is used to enhance prediction for one single meteorological variable, the root mean square error ($RMSE$) as shown as shown in equation (1), is used to be the fitness function for the GA. We refer to this approach as *Single-Variable G-Ensemble*. Referring to equation (1), x_{obs} is an observed value of a variable x and x_{pre} is the predicted one for the same variable.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(x_{obs,i} - x_{pre,i})^2}{n}} \qquad (1)$$

In contrast, as it is necessary to enhance prediction for a set of meteorological variables, the normalized root mean square error ($NRMSE$), is implemented as the fitness function to be minimized during Calibration Phase (equation (2)).

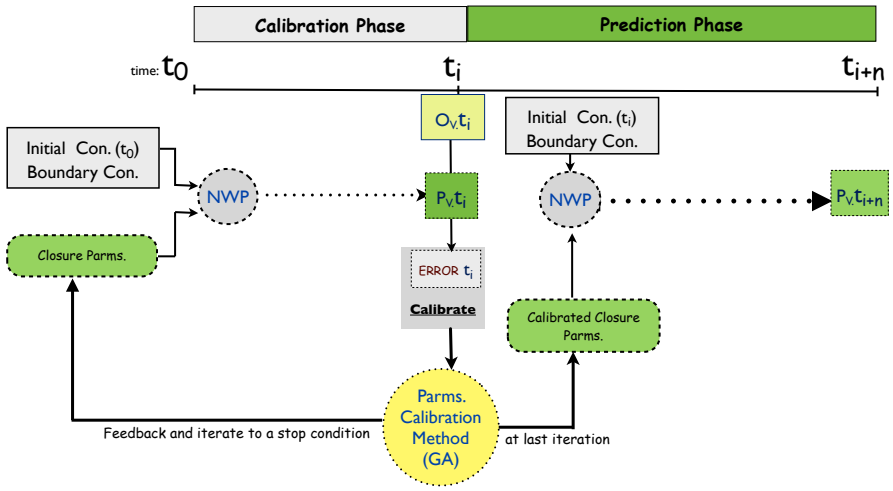


Fig. 1: Two-phase prediction scheme; NWP is the a numerical weather prediction model. t_i is time 00:00 of prediction process, t_0 is a time instant previous to Prediction Phase (initial time of Calibration Phase), t_{i+n} is the future time to be predicted. "O_V" is an observed meteorological variable at time t_i, "P_V" is the predicted variable at the same time using a NWP model.

$$NRMSE = \frac{\sqrt{\frac{\sum_{i=1}^{n}(x_{obs,i}-x_{pre,i})^2}{n}}}{x_{obs(max)} - x_{obs(min)}} \quad (2)$$

This approach is called *Multi-Variable G-Ensemble*. In $NRMSE$ equation, x_{obs} is an observed value of a variable x and x_{pre} is the predicted value for the same variable. The Normalized $RMSE$ ($NRMSE$) is the value of $RMSE$ divided by the range of the observed values of a certain variable. $NRMSE$ indicates the error percentage of the predicted value of a certain variable, compared to the range of its observed values. In order to consider more than one variable at a time, we evaluate $NRMSE$ for all variables, and then, we consider the addition of all of them as the Multi-Variable fitness function.

Despite the fact that the objective in the presented approach is to minimize the $RMSE$ or $NRMSE$ in Calibration Phase, as the fitness function used for the evaluation of ensemble members, other fitness functions can be applied in the presented scheme. The GA could be oriented to minimize any other targeted fitness functions.

At the last iteration in the Calibration Phase, the values of closure parameters, which produced the least value of $RMSE$ or $NRMSE$, i.e. the ensemble member with the best forecast skill score at time t_i, is selected to be used in Prediction Phase. This ensemble member is called: Best Genetic Ensemble Member ($BeGEM$). Our hypothesis is that, for short-range weather forecasts, if the forecast skill is improved in the Calibration Phase by a set of a calibrated closure parameters then the same closure parameter values will also improve forecast skill during Prediction Phase.

By now, in Prediction Phase, a deterministic forecast is used in our experiments. In other words, the $BeGEM$, which is the ensemble member having the calibrated closure parameter values is the single forecast to be conducted in Prediction Phase. However, the produced $BeGEM$ could be integrated in any type of EPS considering perturbations in initial conditions during Prediction Phase.

G-Ensemble scheme was extended in [13] to evaluate ensemble members according to a window of observations rather than 'one-point' observation. Time windowing to the optimization procedure was introduced and the performance (prediction quality) of *G-Ensemble* was enhanced as the used GA was better guided when more observation intervals were considered in the evaluation of ensemble members. Moreover, *Parallel Multi-Level G-Ensemble* was presented in [14], where a multi-chromosome GA was implemented in *G-Ensemble* scheme to optimize various sets of input parameters and the whole scheme was paralleled using Master/Worker paradigm and was tested on a HPC platform.

The obtained results showed significant improvements in prediction quality and less execution times over classical prediction scenarios. It should be mentioned, however, that the implemented GA in *G-Ensemble* scheme was tested in [1], [13], [14] using the same type of GA Crossover and fixed Crossover and Mutation probability ratios. In the next section, more experiments are conducted and discussed regarding different execution scenarios, were different GA configurations are introduced to *G-Ensemble*, in order to evaluate the gained prediction quality in accordance to each different configuration.

5. Experimental Evaluation

To test our approach, we used historical data of hurricane Katrina [15], which occurred on August 28, 2005 in the Gulf of Mexico and unfortunately caused the death of more than 1,800 persons along with a total property damage that was

estimated at $81 billion (2005 USD). The objective of the experiments is to predict the evolution of a meteorological variable from time: 12:00 h. of the day 28/08/2005 to time 00:00 h. of 30/8/2005 (a period of 36 hours in which the major effects of the hurricane were produced). The model is configured to predict the evolution of meteorological variables every three hours; and the spatial resolution of the domain was 12km. The used NWP model in our experiments was WRF and all Physics schemes were the same for all experiments.

To get the evolution of meteorological variables at 12:00 h. of 28/08/2005, we used initial conditions of the atmospheric state in the zone three hours before, i.e. model started prediction from time 09:00 of 28/08/2005. For our approach (*G-Ensemble*), the Calibration Phase started from time 00:00 of 28/08/2005 to time 09:00 of the same day.

The predicted variable in the following experiments is the *Latent Heat Flux (LHF W/m2)* using the *Single-Variable G-Ensemble* approach. However, it must be pointed out that any other meteorological variable could have been used and similar conclusions would be obtained. Examples of such variables can be find in [1], [13], [14], where both approaches of the *G-Ensemble* (*Single-Variable G-Ensemble* and *Multi-Variable G-Ensemble*) are tested to enhance prediction for a set of meteorological variables. The presented results explore the sensitivity of *G-Ensemble* forecast skill to some variations in its GA operations. In particular, we study the sensitivity to the GA Crossover type (one-point and two-points), to the probability to the initial GA population size (initial ensemble members size) and, finally, to the number of GA generation iterations in the Calibration phase.

The goal behind these tests is to provide a more completed insight of the scenarios and possibilities of how to configure an operational *G-Ensemble* according to the time allowed for prediction process and to the number of computing resources available. In the subsequent experiments, prediction errors $RMSE$ produced during Prediction Phase of two ways of prediction are compared:

1) *Single Variable G-Ensemble* approach, with different initial ensemble sizes, Crossover type and ratio, and different number of iterations in Calibration Phase.
2) The EPS approach, which is used to refer to the average error of an ensemble forecast conducted by the initial ensemble members used in the first iteration of Calibration Phase (an ensemble forecast such that the prediction model is different for each ensemble member regarding the targeted parameter values, these variables are not calibrated).

Firstly, Fig. (2) shows an experimental result for a classical EPS prediction of 40 ensemble members (each of which has a different combination of the targeted closure parameters) to predict (every 3 hours) the evolution of *Latent Heat Flux LHF*. The evolution of the values of LHF variable was notably under-estimated in this case. Thus, it could be easily concluded that there is a significant margin of enhancement in prediction which could be achieved.

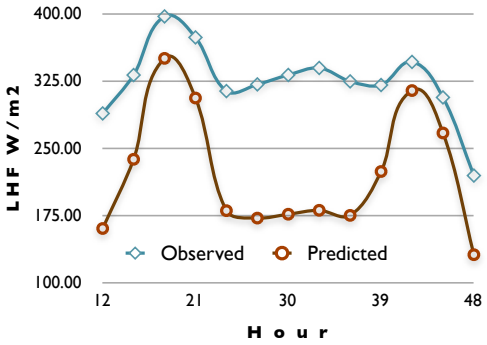
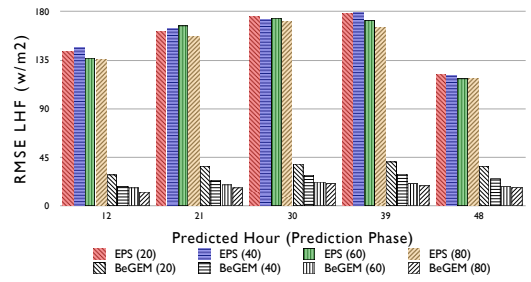
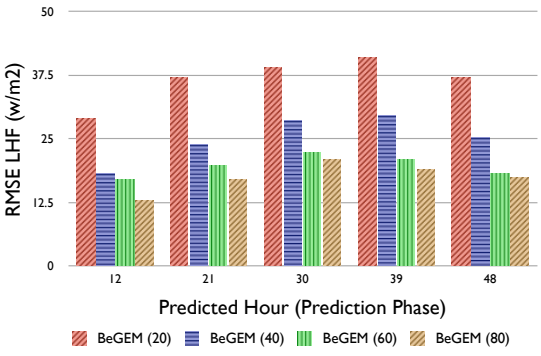


Fig. 2: Classical EPS prediction results compared to observed values.

In Fig.(3(a)), prediction error is shown by using the *G-Ensemble* approach with different initial ensemble sizes to predict LHF variable compared to the classical EPS of the same ensemble sizes. The prediction error of the *G-Ensemble* approach is also depicted alone for the sake of clarity in Fig.(3(b)).



(a)



(b)

Fig. 3: RMSE of LHF pediction. (a): Single-Variable G-Ensemble prediction error Vs. Classical EPS prediction error. Results are of classical EPS(x) and the BeGEM(x), where x refers to the initial ensemble size. (b): A snapshot of (a) to demonstrate RMSE of the different BeGEM(x).

The Genetic Algorithm was configured to iterate 20 times over an initial population size of 40 individuals. Its

three main operators were configured as follows: *Selection*: (elitism: best one of two), *Crossover*: (probability=0.7, type: two points Crossover), and *Mutation*: (probability= 0.2). As shown in Fig.(3), in all cases with different initial ensemble sizes, *G-Ensemble* provides less error values in prediction compared to EPS predictions with the same initial ensemble members. A significant improvement in prediction quality is always gained.

Additionally, it can be observed that increasing the size of an EPS does not produce better results. Actually this happens because EPS results represent an average of the predictions of all ensemble members and, knowing that these members are variated regarding their closure parameters in a random way, using more members does not assure less average error. In contrast, increasing initial ensemble size, which will be calibrated iteratively by the *G-Ensemble* provides better prediction results as observed in the same figure. That is, by increasing the initial ensemble size in *G-Ensemble*, the probability for finding better solutions through GA iterations, also increases.

On the other hand, Fig.(4) shows the GA convergence in the Calibration phase of *G-Ensemble* approach. As such, the error of the best ensemble member through GA iterations is depicted in the figure, using different initial ensemble sizes. As it could be observed, the *BeGEM* produced after 10 iterations when *G-Ensemble* was conducted using an initial ensemble size of 80 members, was equal or slightly better than the same *BeGEM*, produced after 20 iterations when *G-Ensemble* was conducted by 20 initial ensemble members.

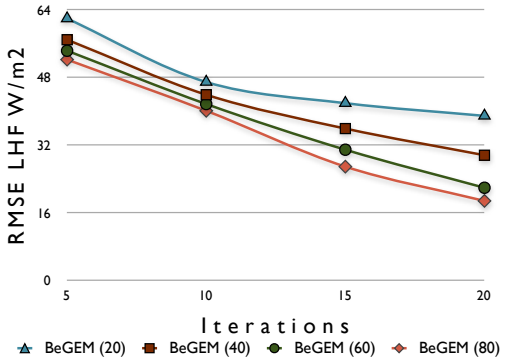
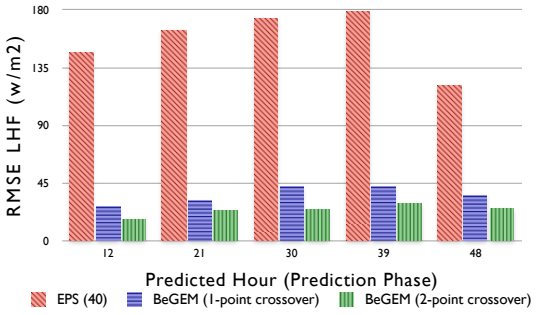


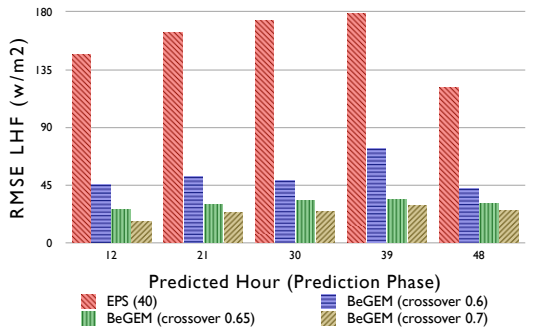
Fig. 4: Calibration phase: BeGEM performance through the Calibration phase iterations for different initial ensemble sizes.

Then, according to the availability of computing resources, their number and the interval of availability, a certain scenario of the combinations between initial ensemble size and number of iterations, could be selected. Execution times and *G-Ensemble* scalability on HPC systems for different combinations regarding the number of iterations during Calibration Phase could be found in [14].

We also tested the *G-Ensemble* approach to predict the same meteorological variable (LHF) by changing the type of the GA Crossover during Calibration phase (Fig. 5(a)), and by changing the GA Crossover probability (Fig. 5(b)).



(a)



(b)

Fig. 5: BeGEM $RMSE$ in prediction of LHF produced in (a): using 1-point and 2-point GA Crossover in the Calibration phase, and (b): using 2-point GA Crossover but with different Crossover probability ratios.

The obtained results show that when *G-Ensemble* used 2-points Crossover in its GA during Calibration phase, prediction results were slightly better, and the same happened when Crossover probability was higher.

That is, when configuring the GA implemented in the *G-Ensemble* scheme on a relatively small size of initial ensemble members, better prediction quality could be obtained by 2-points Crossover and higher Crossover probability. Actually, this is due to the size of the initial ensemble size (initial population size): by using 2-point Crossover and a higher probability of Crossover operations, more variations in ensemble members could be obtained during each iteration of the Calibration Phase. This enhances the ability of the GA to look for better solutions over small initial populations, which is normally the case of NWP executions, where ensemble sizes are normally up to 50 ensemble members.

The results obtained in our experiments confirm our hypophysis that, on one hand, better estimation of model closure parameter values enhances weather prediction quality and, on the other hand, the proposed Calibration Phase leads to better estimation of closure parameter values by tuning the used GA. Additionally, different scenarios could

be applied in an operational *G-Ensemble* according to the available computing resources by variating initial ensemble sizes and the number of GA generation iterations.

We can conclude that *G-Ensemble* is a better choice compared to classical EPS. As shown here, *G-Ensemble* outperforms EPS in terms of parameter estimation but it has been also shown in [1], [13], [14] that the proposed *G-Ensemble* approach is cost effective computationally compared to the classical EPS over a parallel computing environment. In those works many execution scenarios were tested over a HPC environment, and the prediction quality was significantly enhanced, whereas, execution times were reduced in comparison with executions of classical EPS in Prediction Phase.

6. Conclusions and future work

This work describes our ongoing research focused on enhancing short-range weather forecasting by estimating 'optimal' NWP model closure parameter values, using an evolutionary computing method.

In [1], it was shown how forecast skill is sensitive to model closure parameter values. Moreover, *G-Ensemble* prediction scheme was presented, which aggregated a Calibration Phase to the prediction process, where these parameter values were optimized to improve forecast skill. The *G-Ensemble* prediction scheme showed a significant improvement in prediction quality. *Parallel Multi-Level G-Ensemble* was presented in [14], where a multi-chromosome GA was implemented in *G-Ensemble* scheme to optimize various sets of input parameters. Additionally, the whole scheme was paralleled using Master/Worker paradigm and was tested executing it over a HPC platforms. The obtained results showed significant improvements in prediction quality and less execution times over classical prediction scenarios.

In this paper, a complementary work is introduced by conducting and discussing more experiments regarding different *G-Ensemble* execution scenarios, where different GA configurations are introduced to *G-Ensemble* in order to evaluate the gained prediction quality in accordance to each configuration. As a result, it could be concluded that in scenarios of limited number of the available computing resources, where only small ensemble sizes could be applicable, *G-Ensemble* scheme provides better weather predictions by using 2-point Crossover in its GA, and also by using higher Crossover probability ratio. On the other hand, in scenarios where more computing resources are available, and thus, larger ensemble sizes could be used, our results showed that classical EPS does not enhance prediction results by increasing initial ensemble sizes, whereas *G-Ensemble* does. That is, forecast skill in weather predictions could be enhanced almost linearly by *G-Ensemble* scheme as the initial ensemble size increases.

These results encourage us to continue our research efforts by testing our scheme over larger sets of model closure parameters. And we are also planning to design methods that handle real observations during prediction process deciding their injection intervals at run-time in order to get more reliable meteorological predictions.

References

[1] H. Ihshaish, A. Cortes, and M. A. Senar, "Genetic ensemble (G-Ensemble) for meteorological prediction enhancement," in *Proceedings of The 2011 Internacional Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2011).*, H. R. Arabnia, Ed., vol. 1, July 2011, pp. 404–4010.

[2] Weather Research and Forecasting Model homepage. [Online]. Available: http://www.wrf-model.org/index.php

[3] "PSU/NCAR MM5 community model homepage." [Online]. Available: http://www.mmm.ucar.edu/mm5/

[4] T. N. Palmer, "A nonlinear dynamical perspective on model error: A proposal for non-local stochastic-dynamic parametrization in weather and climate prediction models," *Quarterly Journal of the Royal Meteorological Society*, vol. 127, no. 572, pp. 279–304, 2001.

[5] M. Leutbecher and T. N. Palmer, "Ensemble forecasting," *J. Comput. Phys.*, vol. 227, pp. 3515–3539, March 2008.

[6] The ECMWF Ensemble Prediction System homepage. [Online]. Available: http://www.ecmwf.int/research/predictability/projects/index.html

[7] D. J. Stensrud, H. E. Brooks, J. Du, M. S. Tracton, and E. Rogers, "Using ensembles for short-range forecasting," *Monthly Weather Review*, vol. 127, no. 4, pp. 433–446, 1999.

[8] T. N. Palmer and P. D. Williams, "Introduction. stochastic physics and climate modelling," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 366, no. 1875, pp. 2419–2425, 07 2008.

[9] J. Teixeira and C. A. Reynolds, "Stochastic Nature of Physical Parameterizations in Ensemble Prediction: A Stochastic Convection Approach," *Monthly Weather Review*, vol. 136, no. 2, pp. 483–496, Feb. 2008.

[10] P. J. Lawrence and T. N. Chase, "Representing a new MODIS consistent land surface in the community land model (clm 3.0)," *J. Geophys. Res.*, vol. 112, no. G1, 03 2007.

[11] K. W. Oleson, Y. Dai, G. Bonan, M. G. Flanner, E. Kluzek, P. J. Lawrence, S. Levis, S. C. Swenson, and P. E. Thornton, "Technical description of the community land model (CLM)," National Center for Atmospheric Research, Boulder, Boulder, Colo, USA, Tech. Rep. NCAR/TN-461+STR, 2004.

[12] The Community Noah Land-Surface Model homepage. [Online]. Available: http://gcmd.nasa.gov/records/NOAA_NOAH.html

[13] H. Ihshaish, A. Cortes, and M. A. Senar, "Towards improving numerical weather predictions by evolutionary computing techniques," *accepted for publication in: Procedia Computer Science, ELSEVIER*, 2012.

[14] ——, "Parallel multi-level genetic ensemble for numerical weather prediction enhancement," *accepted for publication: Procedia Computer Science, ELSEVIER*, 2012.

[15] Hurricane Katrina homepage. [Online]. Available: http://www.katrina.noaa.gov/

On The Rearrangeability Of Hypercubes Networks Characterization Of Some Non-1-Partitionable Permutations On 4D-hypercubes

Ibrahima Sakho, Jean-Pierre Jung
LITA EA 3097, Université de Lorraine, Metz, France

Abstract - *This paper addresses the problem of the nD-hypercubes interconnection networks rearrangeability that is the capability of such networks to route optimally arbitrary permutations under queueless communication constraints. The k-partitioning is an unifying paradigm of such routing. For a permutation which is not constituted of two sub-permutations on two disjoint (n-1)D-hypercubes of the nD-hypercube, it consists in decomposing it in an upstream permutation routable in k steps and two independent downstream permutations routable in n-k steps on two disjoint (n-1)D-hypercubes. To do so it is essential to know the minimum value of k for which such decomposition is possible, that is the permutation is k-partitionable. This paper characterizes the permutations which can not be 1-partitioned that is which are non-1-partitionable. The characterization alleviates enough the construction of such permutations for low dimensional hypercubes. The paper lists, if they exist, all the classes of such permutations on nD-hypercubes for $n \leq 3$, and some classes for n = 4.*

Keywords: interconnection network; hypercube; permutation; routing; maximum matching of bipartite graph; graph partitioning

1 Introduction

In the research for interconnection networks (IN), hypercubes constitute a very attractive alternative because of their incremental construction which confers them interesting mathematical properties which allow them to match with most of the IN performance criteria.

Several commercial parallel machines [1] have been built over the years and several theoretical and practical research works [2] have also been done on different aspects of their use as IN. Among the theoretical research, one of the most challenging is their rearrangeability that is, their capability to route any permutation in a number of steps that does not exceed the dimension, say n, of the hypercube. In [3], we proposed the k-partitioning paradigm to solve this still-open problem for high dimensional hypercubes. The paradigm consists in decomposing the permutation in an upstream permutation routable in k steps and two independent downstream permutations routable in n-k steps on two disjoint hypercubes. With this purpose, we gave a characterization of non-1-partitionable permutations which allowed proving, at our knowledge for the first time, formally the rearrangeability of hypercubes of dimension n for $n \leq 3$.

This paper gives a new interpretation of this characterization which alleviates enough the construction and the identification of permutations which are not 1-partitionable qualified in the sequel of non-1-partitionable. Indeed, the paper proves more easily that for $n \leq 2$ there is no non-1-partitionable permutation. For n = 3 it lists all the classes of non-1-partitionable permutations. For n = 4 it exhibits all the non-1-partitionability models then lists some classes of non-1-partitionable permutations.

The remainder of the paper is organised in four sections. Section 2 recalls, from [3], the problem formulation and some basic definitions relative to hypercubes, permutations and routing. Section 3 surveys the state of the art on the optimal queueless routing of arbitrary permutations. Section 4 presents the mathematical foundations of the k-partitioning paradigm. Section 5, firstly establishes the new characterization of non-1-partitionable permutations. Then it analyzes, through this new light on the problem, the non-1-partitionability on hypercubes of dimension n = 4. Section 6 concludes the paper.

2 Problem Formulation

2.1 Definitions

2.1.1 n-dimensional hypercube

A n-dimensional hypercube, nD-hypercube, is a graph $H^{(n)} = (V, E)$ where V, the set of the nodes, is the set of 2^n nodes $u = (u_{n-1}u_{n-2} \ldots u_0)$ with $u_i \in \{0, 1\}$ and E, the set of the edges, is the set of the pairs {u, v} of the nodes such that there is one and only one dimension i for which $u_i \neq v_i$.

It comes from this definition that while a 0D-hypercube is reduced to one node, a 1D-hypercube is obtained in interconnecting two 0D-hypercubes, a 2D-hypercube two 1D, a 3D-hypercube two 2D-hypercubes. More generally a nD-hypercube is obtained in interconnecting two (n-1)D-hypercubes. As the interconnection of the (n-1)D-hypercube can be done in anyone of the n dimensions of the resulting hypercube, any nD-hypercube $H^{(n)}$ can be viewed as anyone of the n

couples ($H^{(n)}_{0,\,i}$, $H^{(n)}_{1,\,i}$) of (n-1)D-hypercubes where $H^{(n)}_{x,\,i}$ is obtained by restricting $H^{(n)}$ to its nodes u such that $u_i = x$. Fig. 1 illustrates a 4D-hypercube viewed as the couple ($H^{(4)}_{0,\,3}$, $H^{(4)}_{1,\,3}$).

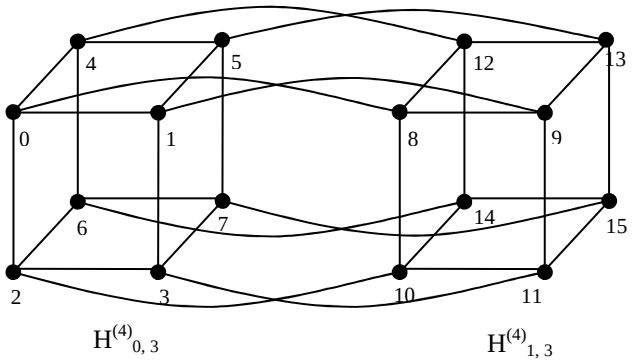


Figure 1. A 4D-hypercube interconnecting two 3D-hypercubes in the dimension 3

2.1.2 Permutation

A permutation on a nD-hypercube $H^{(n)}$ is a one-to-one correspondence π, denoted (π(u); u = 0, 1, ..., 2^n-1), which associates to each node u of $H^{(n)}$ one and only one node of $H^{(n)}$, $\pi(u) = (\pi_{n-1}(u)\pi_{n-2}(u) \ldots \pi_0(u))$.

2.2 Queueless Routing of permutations

Let π be a permutation on a nD-hypercube network with bidirectional links and a set of 2^n messages of the same size each one located at one node u and destined for the node π(u). Routing π under queueless communication constraints consists in conveying all the messages to their respective destination through a sequence of global and synchronous exchanges of messages between neighbour nodes such that no more than one message is located at each node after each exchange step.

Because the messages have the same size, the complexity of such a routing is of the order of the required number of exchange steps. Therefore an optimal routing is the one with the minimal exchange steps. For an arbitrary permutation it is well known, from e-cube routing [4], that at least n exchange steps are required.

3 Related Works

Optimal routing of permutations on nD-hypercubes is, since a quarter of century, one of the most challenging open problems in the theory of IN. It has so been extensively well studied and several communication models and routing paradigms have been used to that purpose.

In [5] Szimansky considers the offline routing in circuit-switched and packet switched commutation under all-port MIMD communication model. Under the circuit-switched hypothesis he proves that, for n ≤ 3, any hypercube is rearrangeable. He also conjectured that routing can be made on the shortest paths, conjecture for which a counterexample has been given in [6] by Lubiw. Under packet-switched hypothesis he shows that routing can be made in 2n-1 steps. Under the single port MIMD communication model, Zhang in [7] proposes a routing in O(n) steps on a spanning tree of the hypercube. In [8, 9] Hwang et al considered online oblivious routing under buffered all port MIMD communication models. They prove that n steps routing is possible for n ≤ 7 and later for n ≤ 12 if local information are used. The better routings under the models viewed above are due to Vöckling [10]. He proves that deterministic offline routing in buffered all port MIMD model can be done in $n+O(\sqrt{n}\log n)$ steps while online oblivious randomized one can be done in $n+O(n/\log n)$ steps.

For the more restrictive models, that is single-port, queueless, and MIMD communication model, the personal communication of Coperman to Ramras, according to Ramaras, and the works of Ramras [11] constitute certainly the leading ones. Indeed while Coperman gives the computational proof that arbitrary permutations can be routed in 3D-hypercube in 3 steps, Ramras proves that if a permutation can be routed in r steps in rD-hypercube, then for n ≥ r arbitrary permutations on nD-hypercubes can be routed in 2n-r steps. Recently, Laing and Krumme in [12] have introduced an approach which simplifies the problem enough to permit a human verification of the possibility of routing in 3 steps arbitrary permutations on 3D-hypercube and computer verification for the 4 steps routing in 4D-hypercube. We also have addressed the problem with a paradigm similar to Laing and Krumme one that we call k-partitioning. However instead of looking explicitly for a partition into 2^k permutations, we look for a partition into two permutations on two disjoint (n-1)D-hypercubes routable in n-k steps. We proved in [3] that for n ≤ 3, arbitrary permutations which are in the worst, 2-partitionable, case can be routed in n steps.

4 Mathematical Foundations

Formally, the k-partitioning process comes down to the computation of a perfect matching in bipartite graphs. So in this section we first recall some basic notions and results on bipartite graphs.

4.1 Bipartite graphs

A bipartite graph is a triplet $G = (V_1, V_2, E)$ where V_1 and V_2 are disjoint sets of nodes and E, the set of the edges, is the set of the pairs {u, v} of the nodes such that $u \in V_1$, $v \in V_2$ and are connected.

The bipartite graph associated to a nD-hypercube is the one where V_1 and V_2 are two disjoint copies of the nD-hypercube nodes and E is the set of the pairs {u, v} such that {u, v} is an edge of the hypercube or u = v.

4.2 Adjacency matrix

The adjacency matrix of a graph is the matrix M whose rows and columns are indexed by the graph nodes and the components M[u, v] are such that M[u, v] = 1 if {u, v}$\in$ E and M[u, v] = 0 otherwise.

For a nD-hypercube, this matrix is anyone of the n (2x2)-blocks matrices whose extra-diagonal blocks, which express the interconnection of its two (n-1)D-hypercubes, are identity matrices and diagonal blocks are the adjacency matrices of its two (n-1)D-hypercubes. Table 1 illustrates the four adjacency matrices of the 4D-hypercubes. In dimension i the indexes of the rows and the columns of the matrix are the boldfaced numbers of the column and the row numbered i in boldfaced italic font.

Table 1. Adjacency matrices of 4D-hypercubes

			3	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
			2	**0**	**1**	**2**	**3**	**8**	**9**	**10**	**11**	**4**	**5**	**6**	**7**	**12**	**13**	**14**	**15**
			1	**0**	**1**	**4**	**5**	**8**	**9**	**12**	**13**	**2**	**3**	**6**	**7**	**10**	**11**	**14**	**15**
3	*2*	*1*	*0*	**0**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**1**	**3**	**5**	**7**	**9**	**11**	**13**	**15**
0	**0**	**0**	**0**	1	1	1		1				1							
1	**1**	**1**	**2**	1	1		1		1				1						
2	**2**	**4**	**4**	1		1	1			1				1					
3	**3**	**5**	**6**		1	1	1				1				1				
4	**8**	**8**	**8**	1				1	1	1						1			
5	**9**	**9**	**10**		1			1	1		1						1		
6	**10**	**12**	**12**			1		1		1	1							1	
7	**11**	**13**	**14**				1		1	1	1								1
8	**4**	**2**	**1**	1								1	1	1		1			
9	**5**	**3**	**3**		1							1	1		1		1		
10	**6**	**6**	**5**			1						1		1	1			1	
11	**7**	**7**	**7**				1						1	1	1				1
12	**12**	**10**	**9**					1				1				1	1	1	
13	**13**	**11**	**11**						1				1			1	1		1
14	**14**	**14**	**13**							1				1		1		1	1
15	**15**	**15**	**15**								1				1		1	1	1

4.3 Matching of a bipartite graph

A matching of a bipartite graph G is a one-to-one correspondence Γ which associates to each node u of a subset of V_1 a node Γ(u) of V_2 such that the set of all the pairs {u, Γ (u)} constitute a set of two-by-two non adjacent edges of E.

A matching is maximal if it admits no other edge without violating the non-adjacency rule of its components.

A matching is maximum (resp. perfect) if its cardinality is maximum (resp. $= |V_1| = |V_2|$).

The computation of a maximum matching is one of the main problems in the study of bipartite graphs. The main results about this computation come from the theorem of Berge [13] whose implementations led to several algorithms.

4.4 k-partitionable permutations

Given a permutation π on a nD-hypercube $H^{(n)}$, 1et r be its optimal routing steps and, for x = 0 and $0 \le i \le n-1$, let:

- $S_{x,i}$ (resp. $D_{x,i}$) be the set of the nodes u of $H^{(n)}$ such that $\pi_i(u)$ (resp. u_i) = x; in fact $D_{x,i}$ is the set of the nodes of $H^{(n)}_{x,i}$,
- $G_{x,i,k}$ be the bipartite graph ($S_{x,i}$, $D_{x,i}$, E^k), k = 0, 1, 2, …, r-1 where $\{u, v\} \in E^k$ if and only if there is a path of length less than or equal k, from $u \in S_{x,i}$, to $v \in D_{x,i}$, and a path of length less than or equal r - k from v to π(u),
- $M_{x,i,k}$ be the adjacency matrix of $G_{x,i,k}$,
- $\Gamma_{x,i,k}$ be a maximum matching of $G_{x,i,k}$.

π is said to be k-partitionable, for k < n, in dimension i if there is a permutation $\Gamma = (\Gamma_{0,i,k}, \Gamma_{1,i,k})$ on $H^{(n)}$ such that α (resp. β) which associates π(u) to Γ(u) such that $\Gamma_i(u) = 0$ (resp. 1) is a permutation on the (n-1)D-hypercube $H^{(n)}_{0,\,i}$ (resp. $H^{(n)}_{1,\,i}$). α and β are then qualified of downstream permutations relatively to Γ which is qualified of upstream permutation.

π is said to be k-partitionable if there is a dimension for which it is k-partitionable. Otherwise it is non-k-partitionable.

Some remarkable k-partitionable permutations are the ones such that $S_{x,i} = D_{x,i}$, which are the only to be 0-partitionable, and the ones such that $S_{x,i} \cap D_{x,i} = \varnothing$ which are 1-partitionable. As a consequence, in the sequel we will

consider only permutations on nD-hypercubes with $n \geq 1$ and which are such that

$$S_{x,i} \cap D_{y,i} \neq \emptyset \text{ for } y = 0, 1.$$

5 Characterization Of Non-1-partitionable Permutations

The k-partionability is the guaranty for a permutation to be optimally routed, first in k steps and then in n-k steps as two distinct permutations on two disjoint (n-1)D-hypercubes. In this intention, it is essential to identify the classes of non-1-partitionable permutations. In [3] we proved the following characterization.

Proposition 1. A permutation is non-1-partitionable in dimension i of a hypercube if and only if one of the adjacency matrices $M_{x,i,1}$ contains a null column.

Let v be a node of the hypercube which corresponds to such a column in the dimension i. Then by construction of of the bipartite graph $G_{x,i,1}$, no of its neighbour nodes, u can belong to $S_{x,i}$. In other words, u can be a source node of no message destined for a node of $D_{x,i}$. Indeed, otherwise, for any such u, by definition, $M_{x,i,1}[u, v]$ should be equal to 1. Therefore any neighbour node u of v is a source node for a node of $H^{(n)}_{\underline{x},i}$ and necessarily, $\pi_i(u) = \underline{x}$. Reciprocally if v is a node of the sub-hypercube $H^{(n)}_{x,i}$ such that $\pi_i(u) = \underline{x}$ for any of its neighbour node u, then the column associated to v in $M_{x,i,1}$ is null and necessarily, from the proposition 1, the permutation which induced $M_{x,i,1}$ is non-1-partitionable in dimension i. It follows:

Proposition 2. A necessary and sufficient condition for a permutation π to be non-1-partitionable in dimension i, is that there is at least a node v of $H^{(n)}_{x,i}$ such that for any node u, neighbour of v,

$$\pi_i(u) = \underline{x}.$$

In the sequel, a node v of $H^{(n)}_{x,i}$ whose neighbours u, including itself, are such that $\pi_i(u) = \underline{x}$ will be called a $\underline{x}$-node.

It is obvious that for $n \leq 2$, there is no non-1-partitionable permutation on nD-hypercubes. Indeed if such a permutation, say π, did exist then according to the node labelling induced by Proposition 2, there would be at least two distinct nodes, say u and v such that

$$\pi(u) = \pi(v)$$

which contradicts the definition of a permutation.

Now let's examine how the characterization from the proposition 2 comes in various forms for $n \geq 3$.

Firstly we will consider the case of the permutations on the 3D-hypercubes and then the one of certain classes of the permutations on the 4D-hypercubes.

5.1 Non-1-partitionable permutations on 3D-hypercubes

Let π be a non-1-partitionable permutation on $H^{(3)}$. Then for any dimension i, there is at least a 1-node, say v, of $H^{(3)}_{0,i}$ such that for any node u neighbour of v,

$$\pi_i(u) = \pi_i(v) = 1.$$

As $H^{(3)}$ is constituted of 8 nodes then necessarily, for any other node w,

$$\pi_i(w) = 0.$$

In particular

$$\pi_i(\underline{v}) = \pi_i(\underline{v} \oplus 2^j) = 0 \text{ for } j = 0, 1, 2$$

where $\oplus$ stands for the bitwise XOR and $\underline{v}$ for the opposite of v obtained by complementing each bit v_i. In other words, $\underline{v}$ behaves similarly to v and therefore is a 0-node. They are the only nodes which behave such that. Furthermore for any node u of $H^{(3)}$,

$$\pi_i(\underline{u}) = \underline{\pi_i(u)}.$$

Fig. 2 illustrates such a situation for i = 0 with node 0 as 1-node.

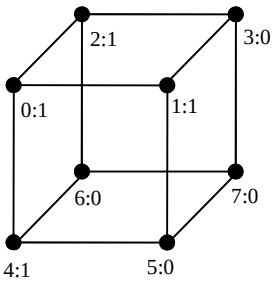


Figure 2. Labelling of the 3D-hypercube nodes according to node 0 as 1-node in dimension 0

Thus, if π is non-1-partitionable on $H^{(3)}$, then for any node u and for any dimension i,

$$\pi_i(\underline{u}) = \underline{\pi_i(u)}.$$

From where

$$\sum_{i=0,2} \pi_i(\underline{u}) * 2^i = \sum_{i=0,2} \underline{\pi_i(u)} * 2^i$$

that is

$$\pi(\underline{u}) = \underline{\pi(u)}.$$

Now let's examine how to choose the x-nodes for each dimension. This is an essential question because any labelling π of the 3D-hypercube such that $\pi(\underline{u}) = \underline{\pi(u)}$ is not a permutation. Indeed, for instance, if the same node is chosen in two different dimensions, then because the neighbour of a node is invariant whatever the dimension,

we obtain 4 nodes whose 2 of the destination addresses bits of the same weight are the same. Consequently whatever the choice for the remaining dimensions the resulting destinations can not lead to a permutation. At least two nodes will have the same destination. Fig. 3 illustrates this case with node 0 as 1-node for both dimensions 0 and 1.

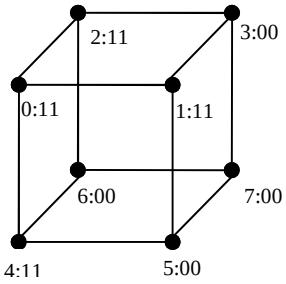


Figure 3. Labelling of the 3D-hypercube nodes according to node 0 as 1-node in both dimensions 0 and 1

So, without loss of generality, we can consider node 0 as the 1-node for the dimension 0. Then, considering the dimension 1, the only choices of 1-nodes are nodes 1, 4 and 5 for which the only choices of 1-nodes for the dimension 2 are respectively 2 and 3, 1 and 2, and 2 and 3; from where the decision tree of Fig. 4 where the numbers of the level dim i stands for the 1-nodes for the dimension i of the hypercube and π^k for the non-1-partitionable permutations classes resulting from each branch.

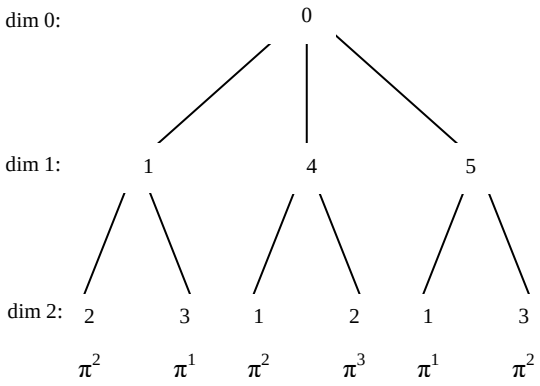


Figure 4. A decision tree for the choice of the 1-nodes in the 3D-hypercube

It can be easily verified that the resulting classes of non-1-partitionable permutations are isomorphic to the following:

$\pi^1 = (3, 7, 5, 6, 1, 2, 0, 4)$,
$\pi^2 = (7, 5, 1, 4, 3, 6, 2, 0)$,
$\pi^3 = (7, 1, 5, 4, 3, 2, 6, 0)$.

Inversely, we can verify that if a permutation is of the class of π^1, π^2 or π^3 then it contains a 1-node for any dimension and then is non-1-partitionable. From where the following proposition:

Proposition 3. A permutation on a 3D-hypercube is non-1-partitionable if and only if it is isomorphic to one of the permutations π^1, π^2, π^3.

5.2 Non-1-partitionable permutations on 4D-hypercubes

Again, let π be a non-1-partitionable permutation on $H^{(4)}$. Then for any dimension i, there is a x-node. On the contrary of $H^{(3)}$, it may exist more than one x-node. Indeed, the labelling of a x-node and its neighbours leaves several alternatives for labelling the remainder nodes. For this reason we have first to determine the maximum number of x-nodes admissible in $H^{(4)}_{\underline{x},i}$ for a given dimension i. Then, in the case of several x-nodes, we have to study the compatibility of all the x-nodes for all the dimensions in the sense that they lead to a permutation and finally to determine the corresponding permutations.

$H^{(4)}_{\underline{x},i}$ admits at most two x-nodes. Indeed the closer the x-nodes are the greater their compatibility is. But the closest three x-nodes are constituted of a sequence of nodes whose binary addresses differ on only one bit which leads to at least 11 destination nodes for 8 available ones. Thus there are two non-1-partitionability models in any dimension: the 1-x-node model and the 2-x-nodes model. In this study we restrict ourselves to the second model. As $H^{(4)}_{\underline{x},i}$ is a 3D-hypercube, three situations may happen according to the distance between the x-nodes: the x-nodes are distant of 1, 2 or 3. Without loss of generality, we can consider the case of the 1-nodes in $H^{(4)}_{0,0}$.

Let node 0 be one of the 1-nodes.

Case 1. The 1-nodes are distant of 1. Let node 2 be the second 1-node. By definition of a 1-node, nodes 0, 2 and all their neighbours, the half of $H^{(4)}$ nodes have to be labelled with 1. Consequently, the other nodes have to be labelled with 0. Fig. 5 illustrates this case.

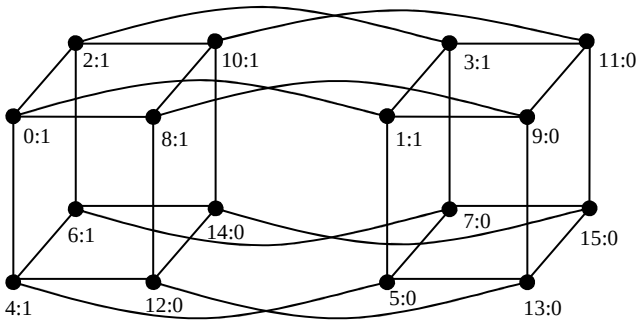


Figure 5. The labelling of $H^{(4)}$ nodes according to the 1-nodes 0 and 2

There is a data flow equilibrium that is a half of the nodes labelled with 1 and the other half with 0. Regarding the queuless constraint, therefore the labelling is consistent. We can observe that the opposites of the 1-nodes in $H^{(4)}$, that is respectively nodes 15 and 13 are 0-nodes. Moreover each x-node in $H^{(4)}$ is a x-node in $H^{(4)}_{x,k}$ where k = 1 is the dimension in which the binary addresses of the x-nodes differ. From the study of 3D-hypercube it comes that in each of the 3D-hypercubes $H^{(4)}_{x,k}$,

$$\pi_i(\underline{u}) = \underline{\pi_i}(\underline{u}).$$

Case 2. The 1-nodes are distant of 2. Let node 10 be the second 1-node. The resulting labelling is illustrated in Fig. 6.

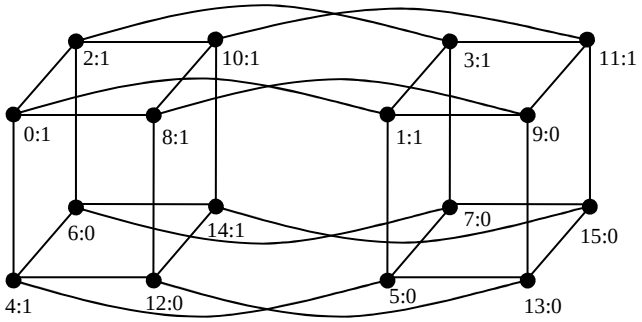


Figure 6. The labelling of $H^{(4)}$ nodes according to the 1-nodes 0 and 10

As previously, the labelling is consistent. We can observe that the nodes 7 and 13 are 0-nodes. Moreover each x-node in $H^{(4)}$ is a x-node in $H^{(4)}_{x,\,k}$ where k=1, 3 is the dimension in which the binary addresses of the x-nodes differ. Again, from the above 3D-hypercubes study, in each of the 3D-hypercubes $H^{(4)}_{x,k}$,

$$\pi_i(\underline{u}) = \underline{\pi_i}(\underline{u}).$$

Case 3. The 1-nodes are distant of 3. Node 0 being one the 1-node in $H^{(4)}_{0,0}$, the only choice for the second 1-node is node 14. The resulting labelling is illustrated in Fig. 7.

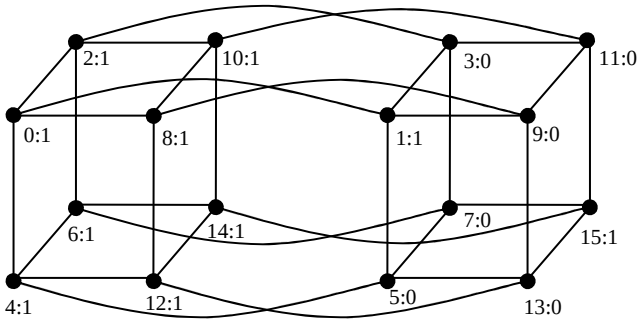


Figure 7. The labelling of $H^{(4)}$ nodes according to the 1-nodes 0 and 14

The labelling is not consistent. There are 10 nodes labelled with 1 against 6 with labelled with 0. This contradicts the fact that π is a permutation on $H^{(4)}$.

This analysis can be summarized by the following proposition.

Proposition 4. If a permutation on $H^{(4)}$ admits two x-nodes in $H^{(4)}_{\underline{x},i}$ then :

a) it also admits two $\underline{x}$-nodes in $H^{(4)}_{x,i}$,
b) both x (resp. $\underline{x}$)-nodes are distant of 1 or 2 in $H^{(4)}_{\underline{x},i}$ (resp. $H^{(4)}_{x,i}$),
c) each x-node remains a x-node in a different $H^{(4)}_{y,k}$ where k is a dimension in which the binary addresses of the x-nodes differ.

In the sequel, we will denote k(i) any dimension k such that each of the two x-nodes for the dimension i remains a x-node in a different 3D-hypercube $H^{(4)}_{y,k}$.

Now we have to study how these two consistent models of non-1-partitionability self combine then how they mingle in the different dimensions of the 4D-hypercubes to lead to a permutation. In this aim, it should be firstly noticed that, similarly to the 3D-hypercubes case and for the same reason, an admissible x-node can not be a x-node in two different dimensions.

First case, suppose that for each dimension the two x-nodes are distant of 1. Then necessarily $k((i) \neq k(j)$ for any $i \neq j$. It can indeed be proved that if, on contrary $k((i) = k(j)$ then in the two remainder dimensions, say r and s, $k((r) = k(s)$. To that purpose, without loss of generality, it suffices to consider that i = 0, j = 1 and that in the dimension 0 the two 1-nodes are the nodes 0 and 2. The decision tree of the remainder 1-nodes choice is illustrated in Fig. 8 where the triplet (u, v):k of the level dim i stands for the two 1-nodes, u and v for the dimension i of the hypercube and k the dimension in which their binary addresses differ.

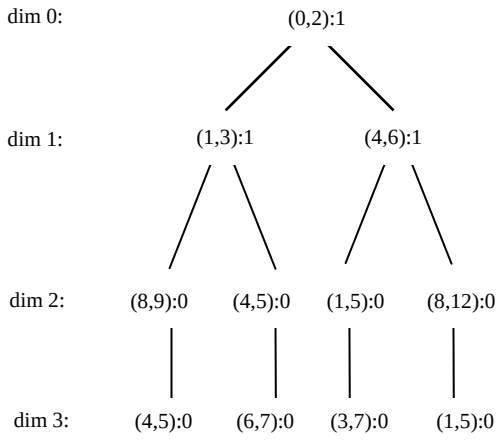


Figure 8. A decision tree of the choice of two 1-nodes distant of 1 in each dimension of the 4D-hypercube

It can be easily verified that no branch of this tree lead to a permutation.

Second case, suppose that for each dimension the two x-nodes are distant of 2. Without loss of generality, let nodes 0 and 10 be the 1-nodes for the dimension 0. Then, in the decision tree of the remainder 1-nodes choice, the only consistent 1-nodes for the dimension 1 are on one side (1,4) which lead to 10 as x-node for a different dimension and on the other side (4, 8) which in turn lead to (1, 2) which lead to 15, the corresponding 0-node of 0 according to Proposition 4, as x-node for a different dimension. In other words, there is no permutation on 4D-hypercube such that there are two x-nodes distant of 2 in each dimension.

Third and last case, suppose that there are two x-nodes distant of 1 for a dimension and two other distant of 2 for a different dimension. Again without loss of generality let 0 and 2 be the 1-nodes for the dimension 0. It can be easily verified that for dimension 1 whatever the x-nodes distant of 2, they lead to a x-node considered for the dimension 0 that is on one side the node 15 as the corresponding 0-node of the 1-node 0 according to Proposition 4 and on the other side the node 2. In other words the x-nodes distant of 1 and the ones distant of 2 can not mingle.

We can conclude this discussion by the fact that the only configurations which could lead to a permutation with two x-nodes in each dimension are the ones with two x-nodes distant of 1 and such that $k(i) \neq k(j)$ for $i \neq j$ for each dimension.

So, let's consider the decision tree of the x-nodes based on these choice criteria. It can then be easily verified that this decision tree leads to permutations which are isomorphic to the following:

π^4 = (3, 11, 13, 15, 1, 10, 9, 8, 7, 6, 5, 14, 0, 2, 4, 12)
π^5 = (7, 15, 9, 13, 3, 14, 11, 10, 5, 4, 1, 12, 2, 6, 0, 8).

The above analysis can then be summarized in the following proposition.

Proposition 6. If a permutation on $H^{(4)}$ admits two x-nodes in each dimension, these x-nodes are distant of 1 and the permutation is isomorphic to π^4 or π^5.

6 Conclusion And Perspectives

This paper has addressed the problem of the characterization of the non-1-partitionable permutations on nD-hypercubes with $n \leq 4$ which is at the heart of the problem of the rearrangeability of the hypercube interconnection networks that is the capability of such networks to realize any permutation without conflict. Non-1-partitionable permutations are the residual permutations which can not be optimally routed under the queueless constraint by the k-partitioning paradigm. So the understanding of their structure is an essential step towards the understanding of how to route them optimally.

We gave a new characterization of the non-1-partitionable permutations which allowed inferring straightforwardly the impossibility of the existence of non-1-partitionable permutations on nD-hypercubes with $n \leq 2$. It also led to easier construction of instances of such permutations on 3D-hypercubes and, to our knowledge for the first time, allows building some classes, the ones with two x-nodes, of non-1-partitionable permutations on 4D-hypercubes without computer simulation.

Our future work will consist in completing this study. It will extend the considered non-1-partitionability models to the 1-x-node ones. Then, beyond the study of how these models self-combine to produce permutations, it will concern the study of how these models mingle consistently with the 2-x-nodes non-1-partitionability models.

7 References

[1] Liu, Y., Han, J., Du, H.: A Hypercube-based Scalable Interconnection Network for Massively Parallel Computing. J. of Comp. 3(10), 58-65 (2008).

[2] Gopalakrishna Kini, N., Sathish Kumar, M., Mruthyunjaya, H.S.: Analysis and Comparison of Torus Embedded Hypercube Scalable Interconnection Network for Parallel Architecture. IJCSNS 9(1), 242-247 (2009).

[3] Jung, JP., Sakho I.: Towards Understanding Optimal MIMD Queueless Routing of Arbitrary Permutations on Hypercubes, Int. J. of Supercomp., DOI: 10.1007/s11227-011-0574-8

[4] Draper, J.T., Ghosh, J.: Multipath e-cube algorithms (MECA) for adaptive wormhole routing and broadcasting in k-ary n-cubes. In: International Parallel Processing Symposium, pp. 407-410. (1992).

[5] Szymanski, T.: On the permutation capability of a circuit switched hypercube. In: International Conference on Parallel Processing, pp. I-103-I110. IEEE Computer Society Press, Silver Spring (1989).

[6] Lubiw, A.: Counter example to a conjecture of Szymanski on hypercube routing. Informations Processing Letters 35, 57-61 (1990).

[7] Zhang, L.: Optimal bounds for matching routing on trees. In: 8th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 445-453, New Orleans (1997).

[8] Hwang, F., Yao, Y., Grammatikakis, M.: A d-move local permutation routing for d-cube. Discrete Applied Mathematics 72, 199-207 (1997).

[9] F. Hwang, Y. Yao, B. Dasgupta. Some permutation routing algorithms for low dimensional hypercubes. Theoretical Computer Science 270, 111-124 (2002).

[10] Vöcking, B.: Almost optimal permutation routing on hypercubes. In: 33rd Annual ACM- Symposium on Theory of Computing, pp. 530-539, ACM Press, (2001)

[11] Ramras, M.: Routing permutations on a graph. Networks 23, 391-398 (1993).

[12] Laing, A. K., Krumme, D. W.: Optimal Permutation Routing for Low-dimensional Hypercubes. Networks 55, 149-167 (2010)

[13] Berge, C.: Graphes. Dunod 3ème édition, Paris (1983.

Performance Analysis of a Matrix Diagonalization Algorithm with Jacobi Method on a Multicore Architecture

Victoria Sanz[12], Armando De Giusti[134], Marcelo Naiouf[14]
[1] III-LIDI, Facultad de Informática, Universidad Nacional de La Plata, La Plata, Buenos Aires, Argentina.
[2] CONICET PhD Scholar [3] CONICET Main Researcher. [4] Full-time Chair Professor.

Abstract - *In this paper, a performance analysis of a matrix diagonalization algorithm with Jacobi method on a multicore architecture is presented. For this, a block-based sequential algorithm was implemented using the CBLAS library, which improves classic sequential algorithms, and then an algorithm for the parallel resolution of the problem with the shared memory programming tool OpenMP is studied. Next, the experimental work is shown and an improvement in response time is observed as more threads/cores are used. Finally, a performance analysis (Speedup and efficiency) as the dimensions of the input matrix and the number of threads/ cores used increase is presented.*

Keywords: multicore architecture, parallel programming, matrix diagonalization, Jacobi, OpenMP.

1 Introduction

The demand for computational power by a large number of scientific applications has increased so much that the use of parallel platforms for their resolution has become essential. Even though *single-core* cluster architectures have become usual due to their cost/performance ratio versus large shared-memory multiprocessor systems, nowadays machines with more than one processor are quite common.

Multicore architectures appear as a response to the limitations for increasing speed in *single-core* processors due to thermal and energy issues. A multicore processor integrates 2 or more cores in a single chip, which means that applications have to be adapted to be able to exploit the parallelism at a thread level provided by this architecture [1]. Similarly, several multicore machines can be connected through a network, which opens the possibility of having clusters with a large number of processors.

The Jacobi method to diagonalize symmetric matrixes has applications in fields such as biometrics, artificial vision, digital signal processing, and so on [2][3][4]. As the volume of input data increases, the computation time required increases significantly. The combination of linear algebra libraries optimized for the underlying architecture, together with the power provided by a multicore, and a suitable parallel programming tool for such architecture, will allow reducing execution time.

Two of the main performance analysis aspects in a parallel system are the *Speedup factor* (*Sp*) [5][6] and the *Efficiency* (*E*) that relates the Speedup with the number of processors (*P*) used [7].

Scalability is a third, very significant factor in parallel applications: problems usually "scale", i.e., the volume of work to be done increases, and the architectures used can also "scale" by increasing the number of processors used. The effect of scaling workload and/or processors on the performance of parallel algorithms, considering *Sp* and *E*, is of interest. A system is said to be scalable if it can maintain a constant efficiency with increasing work and processors [5][8].

The purpose of this work is to implement a parallel algorithm to diagonalize matrixes with Jacobi method, using the shared-memory programming tool OpenMP [9] and the Linear Algebra CBLAS library [10] in order to exploit the computation power of a multicore machine.

Lastly, experimental tests are presented and an analysis of the performance obtained as the size of the matrix and the number of cores scale, is carried out.

This paper is organized as follows: Section 2 includes a description of Jacobi method for real symmetric matrixes; Section 3 presents various implementations of the classic sequential algorithm, two implementations of the block-based algorithm (one using the CBLAS library), and proposes the parallelization of the block-based algorithm using OpenMP; Section 4 includes a comparative analysis of the execution times for the various sequential implementations with matrixes of different sizes, concluding that the block-based algorithm that uses CBLAS performs better, and it presents experimental evidence and an analysis of the performance (speedup, efficiency) obtained with the parallel algorithm as the input matrix size and the number of threads/cores used are increased.

2 Description of Jacobi Diagonalization Method

The problem of diagonalizing a symmetric square matrix S consists in finding an orthogonal matrix X that causes S to be

reduced to the diagonal form. Jacobi method allows finding a matrix X such that: $X^T*S*X = D$.

The elements in the diagonal of matrix D are called eigenvalues of S, and the columns of matrix X are the eigenvectors of S.

2.1. Specific Case: Diagonalization of a 2x2 Symmetric Matrix

If S is a 2x2 symmetric matrix (Fig. 1.a), it can be diagonalized by means of the orthogonal matrix X that is shown in Fig. 1.b. This matrix is known as *rotation matrix*.

$$S = \begin{pmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{pmatrix} \qquad X = \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}$$

a **b**

Fig. 1. a) 2x2 symmetric matrix *S*. **b)** Orthogonal matrix *X*

Since the purpose of the method is obtaining *D*, the values of cosine α and sine α must be such that the elements outside the diagonal of *D* are cancelled out. These values are calculated as shown in Fig. 2.

$$\cos\alpha = \sqrt{\frac{1}{1+\tan^2\alpha}} \qquad \sin\alpha = \cos\alpha\tan\alpha$$

$$\tan\alpha = \frac{sign(\theta)}{|\theta|+\sqrt{1+\theta^2}} \qquad \theta = \frac{s_{22}-s_{11}}{2s_{12}}$$

Fig. 2. Calculation of the values sinα and cosα

2.2 General Case: Diagonalization of an nxn Symmetric Matrix

The general method [11] keeps a matrix X initialized as the identity matrix, where the eigenvectors of S will remain, and in each stage performs a series of rotations that update S and X until the maximum element in the upper triangle of S is lower than a selected threshold.

In each stage, for each element $S_{p,q}$ in the upper triangle, a rotation that cancels out such element is performed. The rotation matrix J (Fig. 3) will have the values cosine α, sine α, -sine α and cosine α at positions $J_{p,p}$, $J_{p,q}$, $J_{q,p}$, and $J_{q,q}$, respectively; it will also have a value of 1 on its diagonal, and 0 on all remaining elements.

The steps to follow for each rotation are:

1. Calculate the elements $S_{p,p}$ and $S_{q,q}$ as follows
   $\mathbf{S_{p,p}} = S'_{p,p} * \cos^2\alpha - 2*S'_{p,q}*\sin\alpha*\cos\alpha + S'_{q,q} * \sin^2\alpha$
   $\mathbf{S_{q,q}} = S'_{p,p} * \sin^2\alpha + 2 *S'_{p,q} *\sin\alpha*\cos\alpha + S'_{q,q} *\cos^2\alpha$
   Notice that $S'_{p,p}$, $S'_{p,q}$ and $S'_{q,q}$ are the values in S before this step. This notation will be used from here on.
2. Cancel out $S_{p,q}$ and $S_{q,p}$.
3. Replace the value in matrix S with the result obtained with the operation J^T*S*J (omitting the positions updated in previous steps).
4. Replace the value in matrix X with the result obtained with the operation X*J.

$$J = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cos\alpha & \cdots & \sin\alpha & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \cdots & -\sin\alpha & \cdots & \cos\alpha & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix}$$

Fig. 3. Rotation matrix J for the general diagonalization case.

Thus, matrixes J_i^T and J_i will be multiplied by S until reaching the diagonal form. Matrix X is the result of the product of the J_i matrixes, and X^T is the result of multiplying the J_i^T matrixes (Fig. 4).

$$\ldots J_3^T [J_2^T [J_1^T \, S \, J_1] J_2] J_3 \ldots = D$$

$$\underbrace{(\cdots J_3^T \; J_2^T \; J_1^T)}_{X^T} S \underbrace{(J_1 \; J_2 \; J_3 \cdots)}_{X} = D$$

Fig. 4. Multiplication of S and the rotation matrixes to obtain the diagonal matrix.

3 Implementations of the Algorithm for Jacobi Diagonalization Method

3.1. Classic Sequential Algorithm:

The algorithm to solve the diagonalization problem does not need to have matrix J stored. The updates carried out in S and X during steps 3 and 4 of the method described in Section 2.2 are replaced by the following operations:

$\mathbf{S_{i,p}} = \mathbf{S_{p,i}} = S'_{i,p*}\cos\alpha - S'_{i,q*}\sin\alpha$ for i=1..n, $i \neq p$, $i \neq q$
$\mathbf{S_{i,q}} = \mathbf{S_{q,i}} = S'_{i,q*}\cos\alpha + S'_{i,p*}\sin\alpha$ for i=1..n, $i \neq p$, $i \neq q$
$\mathbf{X_{i,p}} = X'_{i,p} *\cos\alpha - X'_{i,q}*\sin\alpha$ for i=1..n
$\mathbf{X_{i,q}} = X'_{i,q} *\cos\alpha + X'_{i,p}*\sin\alpha$ for i=1..n

That is, during step 3, rows p and q and columns p and q in matrix S are updated; while during step 4, columns p and q in matrix X are updated.

Two optimizations can be carried out in relation to the storage of matrixes S and X in memory:

- Since S is a symmetric matrix, it is possible to store only n*(n+1)/2 elements, n being the row and columns dimension of S, which allows avoiding unnecessary operations due to symmetry.
- Since access to X is always through its columns, if the elements are stored in memory in column-major order, a better performance will be obtained. This is because every time the memory is accessed to look for an element in X, a block of elements that will be used in the short term is moved to cache memory. This is known as optimization by spatial locality.

3.2 Block-Based Sequential Algorithm:

The development of new architectures requires the implementation of new techniques so that algorithms adapt to the underlying platform and thus obtain a better performance. The classic sequential algorithm described in Section 3.1 can be adapted to use linear algebra optimized libraries.

In order to use the matrix operations available in the Level 3 CBLAS library, a sequential algorithm was implemented for the diagonalization of matrixes using the block-based Jacobi method, which will be the basis for the parallel algorithm; for this reason, the matrix S is stored in full.

This algorithm considers that the dimension of matrix S is n = N*r (Fig. 5), where N is the number of blocks in each dimension, and r is the dimension of each block. This algorithm is similar to the classic one. Each element $S_{P,Q}$, with P,Q=0..N-1, will denote a block. Within the block, the elements will be referenced with indexes (p,q), where p,q = 0..r-1. [12]

$$S = \begin{pmatrix} S_{11} & \cdots & S_{1N} \\ \vdots & \ddots & \vdots \\ S_{N1} & \cdots & S_{NN} \end{pmatrix}$$

Fig. 5. Matrix S, whose dimension is n=N*r.

The method keeps a matrix X initialized as the identity matrix, organized by blocks the same as S, where the eigenvectors of S will remain. Each stage of the algorithm performs a series of rotations that update S and X, until the maximum element in the upper triangle of S is lower than a threshold previously selected.

The operations in any given stage consist in cancelling out each block $S_{P,Q}$ in the upper block triangle in S. The steps to follow for each rotation are:

1. Calculate Jacobi with the classic method over the 2x2-block matrix formed by $S_{P,P}$, $S_{P,Q}$, $S_{Q,P}$ and $S_{Q,Q}$. The eigenvectors will be in a 2x2-block matrix called *auxX*.
2. Calculate the transposition of auxX, *transpX*.
3. Apply rotations to matrix S
   a. $S_{I,P} = S'_{I,P} * auxX_{0,0} + S'_{I,Q} * auxX_{1,0}$ for I=0..N, I ≠P≠Q
   b. $S_{I,Q} = S'_{I,P} * auxX_{0,1} + S'_{I,Q} * auxX_{1,1}$ for I=0..N, I ≠P≠Q
   c. $S_{P,I} = transp_{0,0} * S'_{P,I} + transp_{0,1} * S'_{Q,I}$ for I=0..N, I≠P≠Q
   d. $S_{Q,I} = transp_{1,0} * S'_{P,I} + transp_{1,1} * S'_{Q,I}$ for I=0..N, I≠P≠Q
4. Apply rotations to matrix X
   a. $X_{I,P} = X'_{I,P} * auxX_{0,0} + X'_{I,Q} * auxX_{1,0}$ for I=0..N
   b. $X_{I,Q} = X'_{I,P} * auxX_{0,1} + X'_{I,Q} * auxX_{1,1}$ for I=0..N

3.2.1 Implementation with CBLAS.

In the block-based algorithm, the updates to matrixes S and X (steps 3.a, 3.b, 3.c, 3.d, 4.a, and 4.b) involve a sequence of algebraic operations over rxr-sized blocks or matrixes, and copies of blocks to keep auxiliary data that are not carried out in the classic algorithm. The use of the existing linear algebra libraries allows performing matrix operations (Level 3 CBLAS), achieving the best performance for a given architecture. [10]

The function cblas_dgemm was used to reduce block multiplication and addition processing time, and cblas_dcopy was used to optimize block copying.

3.3. Block-Based Parallel Algorithm with CBLAS:

From Section 3.2, it can be observed that a rotation carried out to cancel out an element $S_{P,Q}$ will update rows and columns with index P and Q in S, and columns P and Q in X. Also, if we have two coordinate pairs (P,Q) and (P',Q'), with P≠Q≠P'≠Q', $J_{P,Q} * J_{P',Q'} = J_{P',Q'} * J_{P,Q}$. Based on these two premises, it is concluded that two or more positions can be canceled out in parallel, provided that their indexes are disjoint [13][14].

The parallelization strategy consists in carrying out the rotations simultaneously in each algorithm stage. The number of parallel tasks to be run in each stage is N/2, since their coordinates must be different, and their indexes will be calculated following the *Chess Tournament* strategy [15]. If there are more tasks than threads, they will be equally distributed.

A task whose coordinate is (P,Q) consists in calculating the eigenvectors and eigenvalues of the 2x2-block submatrix formed by $S_{P,P}$, $S_{P,Q}$, $S_{Q,P}$, $S_{Q,Q}$ with the classic sequential algorithm, independently. Then, matrixes S and X are updated as follows:

- Rotations in S: since there may be several threads in this update stage, they must wait before starting (barrier synchronization); then, each of their columns will be updated (since each thread has coordinates whose indexes are disjoint). Then, after a second barrier synchronization stage, the rows in S are updated. The first synchronization prevents a clash between delayed threads currently updating rows in S and other threads that are about to

update the columns corresponding to the next tasks. The second synchronization prevents conflicts in case one thread is modifying its columns and another thread is updating its rows.
- Rotations in X: these rotations modify only the columns and, since all tasks have different coordinates, they can be done in parallel.

After carrying out these rotations, the threads can move on to complete their following task, and so forth until the current stage is finished. Next, the maximum of the upper triangle of matrix S is calculated in parallel, and then one of the threads calculates the set of indexes for the parallel tasks in the next stage (following the *Chess Tournament* strategy). This last step must be done sequentially because of data dependencies.

For the following step, each thread independently verifies the termination condition, from the maximum value of S obtained before, and, based on the result obtained, the algorithm is ended or a new stage is started.

3.3.1 Implementation with OpenMP.

OpenMP [9] is an API for the C, C++ and Fortran languages that allows writing and running parallel programs using shared memory and offers the possibility of creating a set of threads that work concurrently to exploit the advantages of multicore architectures.

The implementation of the parallel algorithm to solve Jacobi method with OpenMP loads the input matrix S and sequentially initializes matrix X.

Next, a set of threads is created, as many as cores are available in the multicore architecture, and they initialize in parallel the structure with the indexes of the tasks to be run in the first stage.

In each stage of the algorithm, each thread will take a consecutive set of parallel task indexes and will follow the steps described in Section 3.3. When all the tasks in a stage have been executed, the threads calculate in parallel the maximum of the upper triangle in S by applying a reduction operation. Then, one of them updates the set of parallel task indexes for the next stage, following the *Chess Tournament* strategy.

4 Results

Tests were carried out in a machine with two Intel Quad Core Xeon E5405, 2.0Ghz processors. Each core has a 32-Kb L1 cache for data and a 32-Kb cache for instructions. Each pair of cores shares a 6-Mb L2 cache. Available memory is 10Gb RAM.

All times were measured with the function omp_get_wtime from the omp.h library. The tests that are compared in Sections 4.1, 4.2 and 4.3 were carried out over matrixes with identical data and identical precision (0.0001), since convergence times depend on the input data and the precision selected. The same is valid for Section 4.4, which assesses the scalability of the parallel algorithm.

4.1. Classic Sequential Algorithm

Four versions with different methods for storing matrix S and matrix X were implemented:

- Version 1: it stores the entire S by rows and X by rows
- Version 2: it stores the entire S by rows and X by columns. Optimization by spatial locality in X.
- Version 3: it stores only the upper triangle for S, and it stores X by rows. Storage optimization for S and lower number of operations by symmetry.
- Version 4: it stores only the upper triangle for S, and it stores X by columns.

Table 1 shows the execution times for each of the versions of the classic algorithm, and matrixes with n=100,200,300,400,500,600,700,800,900,1000. Versions 2 and 3 improve times compared to version 1, the optimization implemented for version 3 being the one with the greatest impact. For this reason, when both optimizations are incorporated in version 4, execution times are reduced even more.

Table 1. Sequential algorithm times with classic Jacobi for various optimizations.

Dimension / Version	Version 1	Version 2	Version 3	Version 4
100	0.21	0.2	0.18	0.18
200	2.1	1.92	1.78	1.62
300	7.56	6.77	6.17	5.54
400	20.51	18.58	17.37	14.97
500	40.8	36.72	34.7	29.57
600	72.4	64.44	60.7	52.4
700	120.7	104.34	98.56	84.2
800	187.3	159.41	151.9	127.4
900	307.03	258.07	250.8	205
1000	387.52	323.5	318.5	256.42

4.2 Block-Based Sequential Algorithm

For the tests with block-based sequential algorithms, the same matrixes from Section 4.1 were used, but their data were organized in consecutive blocks whose possible size will depend on the size of the matrix.

4.2.1 Block-Based Sequential Algorithm (without CBLAS)

The block-based algorithm that does not use CBLAS did not improve the times obtained by the classic algorithm version 4. As the tests included in Table 2 show, the best

response time is obtained with the maximum possible block size. In those cases, the block-based algorithm behaves very similar to the classic algorithm, since it carries out only one stage to obtain the eigenvalues and eigenvectors of the 2x2-block matrix formed by $S_{0,0}$, $S_{0,1}$, $S_{1,0}$ and $S_{1,1}$ applying the classic algorithm, obtaining in this way the eigenvalues and eigenvectors of the entire matrix.

The reason for which the block-based algorithm without CBLAS did not improve response times is due to the extra time used to multiply and copy blocks, and as block size decreases, a larger number of these operations is required.

4.2.2 Block-Based Sequential Algorithm with CBlas

The block-based sequential algorithm that uses the CBLAS library improved the response times of the sequential algorithms assessed in the previous sections. In most of the cases, the best time was obtained with a block size of 10, as shown in Table 3.

To analyze the cause of this improvement, the number of stages carried out by the block-based algorithm in its two versions and the average time needed to perform a rotation were calculated. These tests were done for the matrix of size 1000 with block size 10 and 500.

Both algorithms performed an only stage for block size 500, and 10 stages when using the block size 10. This proves that, when block size is reduced, the algorithm will need more iterations to converge. However, the average time needed for each rotation is different. Taking into account that in each stage, N*(N+1)/2 rotations are done, it is observed that:

- In the case of the block-based algorithm without CBLAS, the average rotation time for the matrix with block-size 10 is 0.011183. The number of rotations in each stage is 5050. Therefore, each stage will take approximately 56.4 seconds, which explains the time measured during the test. On the other hand, the average rotation time for the matrix with block-size 500 is 386.74. This time is similar to the final time obtained, and also to the time obtained with the classic algorithm (version 1). This is because there is only one stage and a single rotation.

- In the case of the block-based algorithm with CBLAS, when using block-size 10, the average rotation time is 0.00229; since there are 5050 rotations in each stage, each will take approximately 11.56 seconds, which explains the time measured during the test. When using block- size 500, the iteration time was 370.81.

The improvement achieved with the block-based algorithm that uses CBlas is due to the use of the functions cblas_dgemm and cblas_dcopy when calculating rotations.

The size of the block that optimizes times must be such that calculating the classic Jacobi for a 2x2-block matrix is not so expensive, since times that are similar to those of version 1 of the classic algorithm will be obtained. The exact ratio must be found, since very small blocks will reduce the average rotation time but will increase the number of rotations. For example: for the block-based algorithm with CBLAS, the average rotation time for the test with the matrix of size 1000 and block-size 5 is 0.001209. The algorithm carried out 10 stages and 20100 rotations in each stage. Each stage in average lasted 24.3 seconds, which ended up in an increase in time.

Table 2. Times obtained with the block-based sequential algorithm without CBLAS.

Dimension / Block Size	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**
5	0.5	4.29	15.92	37.54	72.87	129.72	198.97	337.1	468.61	578.75
10	0.52	4.05	14.46	33.27	65.12	108.82	172.14	282.6	401.9	549.71
25	0.57	4.78	14.66	36.09	67.01	121.98	189.6	278.54	428.09	530.05
50	0.21	4.94	18.7	41.81	82.4	135.84	224.11	324.8	446.6	650.55
75			18.8							
100		2.03		47.9		172.2		405.77		795.07
125					97.84					818.09
150			7.19			172.18			632.6	
175							319.66			
200				19.94				485.77		
225									702.07	
250					39.9					1004
300						69.65				
350							111.99			
400								194.47		
450									270.99	
500										387.25

Table 3. Times obtained with the block-based sequential algorithm with CBLAS.

Dimension / Block Size	100	200	300	400	500	600	700	800	900	1000
5	0.22	1.83	5.68	14.58	30.86	48.12	75.2	111.45	175.6	241.5
10	0.26	1.47	4.25	8.83	15.75	25.38	41.9	60.6	83.83	112.9
25	0.44	2.68	6.86	14.17	24.5	36.35	51.77	70.77	94.12	119.7
50	0.2	3.82	12.35	25.19	43.46	65.83	97.2	131.54	169.9	226.18
75			14.76							
100		1.95		38.37		118.44		247.5		427.22
125					78.36					508.5
150			6.87			153.86			446.86	
175							258.65			
200				19.34				390.27		
225									581.52	
250					38.48					811.84
300						67.05				
350							108.62			
400								189.45		
450									259.3	
500										371.03

Table 4. Times obtained with the block-based sequential algorithm that uses CBLAS for matrixes whose n is power of 2.

	Matrix Dimension					
Block Size	**256**	**512**	**1024**	**2048**	**4096**	**8192**
2	25.76	220.76	2070.03	17403.73		
4	4.86	38.54	364.5	3011.32		
8	2.59	17.28	136.36	1100.64	8685.8	
16	3	17.78	106.14	686.25	5097.1	39495.26
32	4.98	27.9	156.73	862.54	5493.24	37061.84
64	7.01	47.02	259.42	1254.09		30345.76
128	4.05	69.09	462.23	2448.01		53193.4
256		40.16	687.7	4590.44		
512			1483.56	26982.67		
1024				12963.09		

4.4 Block-Based Parallel Algorithm with CBLAS

Since the scalability analysis was carried out with 2, 4, and 8 threads/cores, matrixes whose size is power of 2 were used, so that the tasks were proportionally distributed among the threads.

Table 4 shows the execution times of the block-based sequential algorithm that uses CBLAS for those matrixes, varying the size of the blocks within the possible values. The test that optimized the execution time for each matrix size is highlighted. Table 5 shows the speedup (*Sp*= Sequential Time / Parallel Time) and the efficiency obtained (*E*= *Sp* / Number of processors) in the parallel tests.

These results show that, if matrix size is kept the same and the number of processors is increased, the speedup obtained is better, that is, the problem is solved in less time. This improvement, however, does not keep a constant efficiency. The decrease in efficiency is due to the overhead of thread creation, synchronization (barriers), and the sequential portion of the problem (the structure with indexes of parallel tasks must be updated in each stage following a *Chess Tournament* strategy in a sequential manner).

When scaling the problem and keeping the same number of processors, as shown in Table 5 and Fig. 6, both speedup and efficiency improve in general, since the overhead mentioned above is less significant in the total processing time.

Table 5. Speedup and Efficiency values as matrix size and threads/cores are increased.

	Speedup			Efficiency		
Dimension (r) / Parallel Tasks	**2**	**4**	**8**	**2**	**4**	**8**
256(8) – 16	1.65	2.91	5.08	0.82	0.73	0.63
512(8) – 32	1.62	2.80	4.91	0.81	0.70	0.61
1024(16) – 32	1.88	3.27	6.30	0.94	0.82	0.79
2048(16) – 64	2.00	3.68	6.65	1.00	0.92	0.83
4096(16) – 128	1.98	3.82	7.07	0.99	0.96	0.88
8192(64) – 64	1.92	3.72	7.07	0.96	0.93	0.88

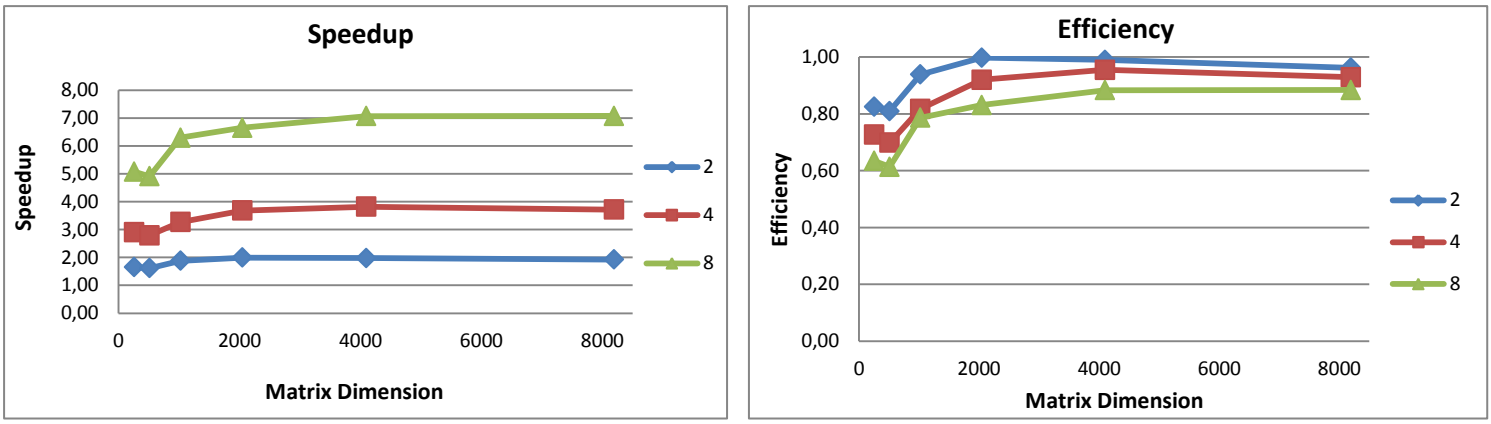


Fig. 6. a) Speedup as matrix size and the number of threads/cores are increased **b)** Efficiency as matrix size and the number of threads/cores are increased.

The behavior described is typical of a scalable parallel system, where efficiency can be maintained at a constant value by simultaneously increasing the number of cores/processors and the size of the problem.

5 Conclusions and Future Work

Various sequential algorithms for the resolution of the matrix diagonalization problem were presented and one parallel implementation that exploits the power offered by multicore architectures was introduced.

The resulting execution times were analyzed for each implementation, and it was observed that the best performance corresponds to the algorithms that use libraries optimized for linear algebra (CBLAS Level 3). It was also observed that performance improves when the algorithm is parallelized on a multicore architecture.

In recent years, GPUs (Graphic Processing Unit) [16] have gained significance due to the high performance achieved in general-purpose applications. One of the future lines of work is based on the migration of Jacobi diagonalization algorithm to be run on GPU, and then systematically study the performance achieved as the size of the problem and the number of threads are increased. Also, an energy consumption analysis for the execution of this parallel algorithm on various multicore architectures is proposed [17].

6 References

[1] Olukotun K, Hammond L, Laudon J. Chip Multiprocesor Architecture. Synthesis Lecture on Computer Science, Morgan&Claypool; 2007.
[2] Turk M., Pentland A. Eigenfaces for recognition. Journal of Cognitive Neuroscience Vol 3, No. 1, pp- 71–86; 1991.
[3] Bravo Muñoz I. Arquitectura basada en FPGAs para la detección de objetos en movimiento, utilizando visión computacional y técnicas PCA. Doctoral Dissertation, Universidad de Alcalá; 2007.
[4] Brent R., Parallel Algorithms for Digital Signal Processing. Numerical Linear Algebra. Digital Signal Processing and Parallel Algorithms, pp. 93-110. Springer, Heidelberg; 1991.
[5] Grama A., Gupta A., Karypis G., Kumar V. Introduction to Parallel Computing. Second Edition. Addison Wesley; 2003.
[6] Leopold C. Parallel and distributed computing. A survey of models, paradigms, and approaches. Wiley; 2001.
[7] Quiin M. J. Parallel Computing: Theory and Practice. McGraw-Hill Companies; 1993.
[8] Hwang K. Advanced Computer Architecture. Parallelism, Scalability, Programmability. McGraw Hill; 1993.
[9] The OpenMP API specification. http://openmp.org/wp/.
[10] BLAS Basic Linear Algebra Subprograms. http://www.netlib.org/blas/
[11] Rutishauser H. The Jacobi Method for Real Symmetric Matrices. Numer. Math. 9, 1-10; 1966.
[12] Hansen, E. On Jacobi methods and block-jacobi methods for computing matrix eigenvalues. Doctoral Dissertation, Stanford; 1960
[13] Sameh A. On Jacobi and Jacobi like algorithms for a parallel computer. Mathematics of computation, Vol 25, No. 115, 1971.
[14] Shroff G. A parallel algorithm for the eingenvalues and eigenvectors of a general complex matrix. Research Institute for Advanced Computer Science. NASA Ames Research Center. Technical Report.; 1989
[15] Bischof C., Van Loan C. Computing the singular value decomposition on a ring of array processors. Proceedings of the IBM Europe Institute Workshop on Large Scale Eigenvalue Problems. Vol. 127, pp 51–66; 1986.
[16] General-Purpose Computation on Graphics Hardware http://gpgpu.org/.
[17] Wu-chun Feng, Xizhou Feng & Rong Ge. Green Supercomputing Comes of Age – Journal IT Professional. Vol. 10, No. 1, pp 17-23; 2008.

Shared Memory Computing with Virtualized PCI-e IO and Addressing

The 2012 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)

Ayman G. Fayoumi, King Abdulaziz Univ.

Forrest K. Blair, Virtuon Inc.

Joseph Y. Hui, King Abdulaziz Univ., Arizona State University, and Virtuon, Inc.

Haojun Luo Arizona State University

Patrick Martin, Arizona State University

***Abstract*— For byte memory addressing, we propose using a Memory Area Network for the purpose of multiple processor access to unlimited byte addressed solid state memory on a local, or perhaps a wide area network. We categorize various shared memory computing architectures. We then introduce iPCI to provide the networking and IO virtualization method for shared memory computing. A novel PCI-e switch on a chip with multiple Terabit per second throughput is described. Applications of this shared memory computing approach are explored.**

Keywords- Virtual Shared Memory; IO Virtualization; Memory Area Network; Terabit Switching; Memory Allocation

I. INTRODUCTION TO SHARED MEMORY COMPUTING AND MEMORY AREA NETWORKS

High Performance Computing (HPC) has undergone significant changes, from initial focus of the 1980s of supercomputing based either on ultrafast or massively parallel computing, to network/grid computing of the 1990s based on broadband Internet, to data-centric computing of the 2000s based on data mining methodologies. In recent years, a new paradigm of cloud computing has dominated both computer science research and the information technology industry.

The key technological foundation of cloud computing is virtualization of physical resources in the cloud, including servers, desktop computers, data storage, computer memory, computer IO [1], data networks, and application software. A focus of our research is on memory virtualization in the cloud [2], in short cloud memory, which we believe to be significant for scientific computing in a paradigm called shared memory computing or memory area networks (MeMAN).

Virtual memory has been a foundation of computer engineering from the early days when operating systems were developed around the management of computer memory. Hypervisors, essentially cloud based operating systems, can allow byte address memory space anywhere in the cloud to be allocated to physical machines (PM), virtual machines (VM), or application software compiled and executed anywhere in the cloud.

The intellectual framework of cloud memory is the deconstruction of the physical computer and associated operating system, so that the cloud now comprises a wide area memory network shared and addressed globally for purposes such as data mining. Data-structures then become global, byte-addressed by any program or user in the cloud. This paradigm is sometimes referred to as shared memory computing (SMC). SMC is an HPC method, a competitor to methods of massively parallel computing (MPC) based on message passing among processors.

SMC has its limitation due to latency and throughput impairments in the cloud. Network protocols, such as the Internet Protocol (IP) are unreliable and slow, and therefore fail to meet the nanosecond requirements of computer memory read and write. Cloud memory faces a similar problem that was encountered by data storage, such as NFS, SAN (storage area network), and iSCSI (internet SCSI), only the response time is orders of magnitude faster (tens of nanoseconds instead of tens of microseconds for SAN).

Therefore significant research is needed to first classify types of cloud memory and second to invent high performance cloud memory, a focus of our research.

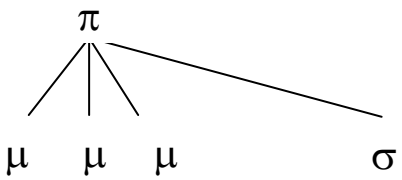
II. CLASSIFICATON AND USE OF SHARED MEMORY COMPUTING

Classification of Memory Sharing

Various classification of memory sharing is proposed here from the simplest to the most general and powerful. Using Greek nomenclatures, processors are represented as π, memory as μ, storage as σ, and bridges/switches as β.

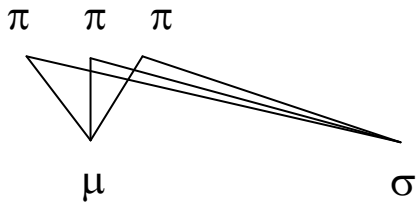
One-to-Many Access

In the simplest form, we have one processor that is directly attached to a multiplicity of memory banks, achieving a scaling of memory capacity for a single powerful processor as shown below. In storage technology, this is analogous to direct attached storage (DAS).



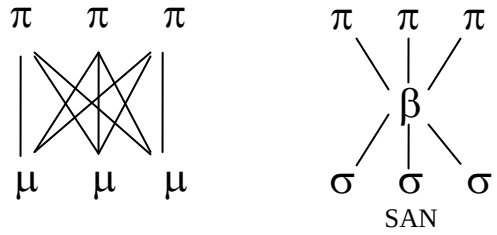
Many-to-One Sharing

Multiple processors share a large memory for the purpose of effective allocation of memory on demand as shown following. This is analogous to the concept of network attached storage (NAS) such as used for network file servers (NFS) accessing a shared data store. However, access to the memory can only be one file or block at a time by servers.



Many-to-Many Sharing

In this more general scheme, multiple processors can access multiple memory banks simultaneously as shown in the left figure in the following. This is analogous to the concept of storage area network (SAN), in which multiple storage servers can serve data retrieval requests on the Internet, while these servers can access simultaneously multiple storage banks through a dedicated switching fabric of a SAN as shown in right figure in the following. Performance is high because of dedicated hardware based protocol processing, such as the use of the Fiber Channel (FC) protocol.



SAN

Exclusive versus Shared Memory Allocation

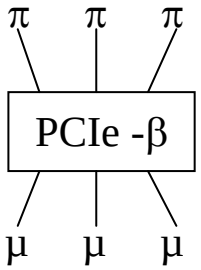
In the above discussions, it is assumed that PM or VM uses memory location exclusively. This is the traditional role of an operating system allocating memory to devices. At the software or program level, a compiler can allocate memory (e.g. the malloc procedure call in C) that can be shared by different procedures or objects in programs. The allocation of memory in this case is facilitated by a compiler prior to further allocation by the operating system. Memory locking mechanisms such as semaphores may be required to ensure memory consistency in conflicting memory access. Examples of research on shared memory computing are shown in references [6-14]

Protocol Adaptation

Virtual memory allocation in the cloud requires IO virtualization, also in the cloud. This is analogous to virtual storage allocation by the iSCSI protocol, whereby the physical SCSI bus for disk drives is extended by the Internet (hence the term Internet SCSI). In the case of cloud or networked virtual memory, the prevailing memory access IO mechanism nowadays is the PCIe (Peripheral Computer Interface) bus. In our previous patented research efforts, we invented various techniques to extend the PCIe bus, for example via wireless networking, via Ethernet switching, via optical networks, and most extensively via the Internet. The choice of protocol for which PCIe adapts to depends on the geographic extent of the cloud, as well as the latency and throughput requirements for cloud memory.

Memory Switching

Many-to-many cloud memory requires a memory switch amongst processors and memories. The memory switch allows highly parallel access of memory banks by processors. Since memory banks and processors can be extensively connected by PCIe bridges, our research focuses on building large PCIe bridges. In previous patented research, we have invented a Terabit switch on a chip with no queuing delay [3, 4, 5]. We are extending the research into memory switching applications. The memory area network (MeMAN) [2] architecture is shown in the following figure.



MeMAN
(Memory Area Network)

III. VIRTUALIZATION OF PCI-E AS A FORM OF SMC

PCI Express (PCI-e) has the ability to be virtualized using the PCI Express relay over an Ethernet channel that uses MAC addresses and Ethernet switches. PCI Express supports the following transaction types: memory, I/O, and data configurations. The virtualization can be implemented by using a native PCI Express host bus adapter on a computer and a scalable network (preferably 1-10 Gbps or higher). In terms of shared memory computing (SMC), the PCI Express architecture is ideal due to the large density of computers and storage.

The PCI Express can be broken down into three separate layers: transaction layer, data link layer, and physical layer. The transaction layer deals with application data. In PCI-e virtualization, the transaction layer can be virtualized through

packet based protocols known as TLPs (Transaction Layer Packets) [23]. A header can be added to the TLP which creates the exchange of protocol specific configuration and synchronization information between the host bus adapter (HBA) and the remote bus adapter (RBA). The resulting PCI-e transaction layer packet is then encapsulated within an Ethernet frame [22].

The data link layer's primary function is to handle the communication protocols of acknowledge (ACK) and negative acknowledge (NAK) supplied from the transaction layer [23]. The data link layer can be translated into an Ethernet implementation of the protocol data units known as DLLPs (Data Link Layer Packets). When data transactions are being processed over an Ethernet switched network, the transaction layer packets are verified for data integrity by the data link layer logic of the receiving end. The proprietary receiving port responds with a either an ACK or NAK, which is then passed to the PCI-e protocol logic. After, a MAC address is then generated that corresponds to the source. The MAC address is parsed with the protocol data to the MAC core. Data integrity is ensured for DLLPs by using a cyclical redundancy check (CRC). A CRC is included with each packet sent across the link [22]. The following are the two plausible cases:

Case 1: ACK is received. The native MAC core transmits a Link ACK packet that informs the receiving port of the next packet what to look for. The MAC core at the other end of the link, receives a Link ACK packet and passes it to the PCI-e DLLP handler. The PCI-e DLL handler that received the Link ACK packet translates the packet into a PCI-e ACK packet and passes it back to the original sending port. The port then processes the ACK and removes the associated TLPs from its replay buffer due to a successful transmission [22].

Case 2: NAK is received. The native MAC core transmits a Link NAK packet stating that the last packet of the receiving packet was received successfully. The remote MAC core then receives a Link NAK packet which is passed to the PCI-e DLLP handler. The PCI-e DLLP handler receives the Link NAK packet and translates it into a PCI-e NAK packet, passing it to the original sending port. The port processes the NAK and resends the outstanding TLP(s) in the replay buffer [22].

The physical layer processes the packets that arrive from the data link layer. The physical layer converts the packets into a serial bit stream. The physical layer connects to the motherboard of the computer. The link created is referred to as a PCI Express Link. The PCI Express Link communicates with the root bus of the native host computer. A PCI Express Link must consist of lanes in parallel (i.e. x1, x4, x8, x16, etc.). The physical layer can be divided into two sub-blocks: a logical and an electrical sub-block. The logical sub-block has a Receive section that identifies and prepares any received data before it is passed to the data link layer. The electrical sub-block has a Transmit section that prepares outgoing data from the data link layer for the electrical sub-block [24].

IV. A TERABIT PCI-E SWITCH FOR SMC FABRIC

In this paper, we also present a large number of input/output ports on the PCIe switch ASIC for SMC applications. The switch is designed as a 128x128 Clos-network crossbar switch which can provide a throughout of 1.024Tb/s for PCIe 2.0 system [24]. Instead of switching at Transaction Layer, PCIe signals are transmitted as is through the crossbar fabric. Without regeneration and buffering of PCIe packets, much of the processing and delay is removed. We also propose a new output contention resolution algorithm utilizing an out-of-band protocol to control the crossbar switch before sending PCIe packets. The proposed switch becomes a switching physical medium for PCIe packets.

In the proposed switching system, electrical PCIe signals are transmitted as is through a three-stage Clos-network crossbar switch called Physical Plane (PP). As a result, without regeneration and buffering of PCIe signal and data, much of the processing and delay is removed within the switch fabric. This requires a shift of accessing, routing and other functions to the endpoints. We employ contention resolution in both space and time to route data through the Physical Plane. The endpoint interface translates the destination PCIe address into routing address. The routing address is sent prior to sending PCIe packets.

The block diagram of the proposed PCIe Switch Board (SB) is shown is Figure 1. PCIe endpoint named Host Bus Adapter (HBA) that generates PCIe signals are connected to the proposed switch ASIC via PCB traces. Each port has 2 pairs of traces for which one pair is responding for transmitting and the other is for receiving. As an illustration in Figure 1, 2 pairs of wires carrying Low-Voltage Differential Signaling (LVDS) for PCIe 2.0 [25] are switched in a Physical Plane (PP). The HBA, Control Plane and Physical Plane are connected through an interface named CC/PP Interface which provides different connections among these components.

128 HBAs are numbered sequentially in binary representation, i.e. the inputs are numbered in the figure as 0000000 to 1111111, respectively. This number is referred to as an Input Segment Address (ISA).

A three-stage Clos-network is built in the PP with a total number of 24 16x16 sub-crossbar switches named Switch Plane (SP). Each stage has 8 Switch Planes numbered as 000 to 111. The structure of Physical Plane and 16x16 Switch Plane is shown in Figure 2. HBA transmitter side (TX+/-) is connected to receiver side (RX+/-) through the three-stage Clos-network switch. It can provide alternative end-to-end connections by routing in different Stage 2 SPs. In each 16x16 SP, X-Y based crossbar switch is structured. The connection between two ports is established using a CMOS Transmission Gate (TG) with low on-channel resistance. The connection status of TG is controlled by the Control Lines.

The step-by-step and out-of-band control architecture allows the initiating endpoint to choose alternative paths in the second stage switch planes to avoid blocking. Previous work [26-27] has been done to analyze the throughput of the proposed switching system.

To verify the proposed switching concept and evaluate performance, we designed the layout of the 16x16 crossbar switch using TSMC 0.25μm CMOS process. With a total number of 256 NMOS transistors used as crosspoints, the layout size is 0.54mm x 0.74mm. Post-layout simulation is

preformed to verify the signal integrity of transmitting PCIe signals.

For the three-stage switch fabric, an approximate area of 5.4mm x 2.4mm is needed. Counting the size of Control Plane, it is not difficult to fabricate the proposed switching system on a single chip with today's ASIC packaging technology.

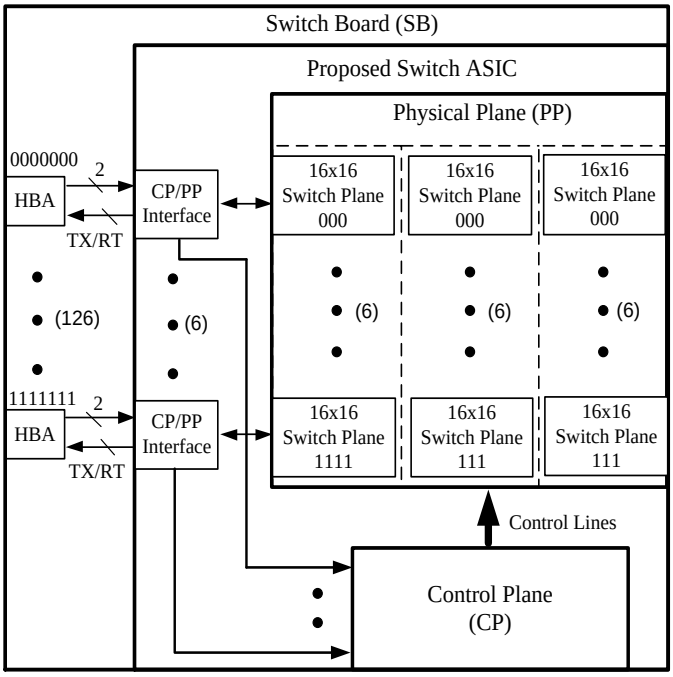


Figure 1. The block diagram of the proposed switching system

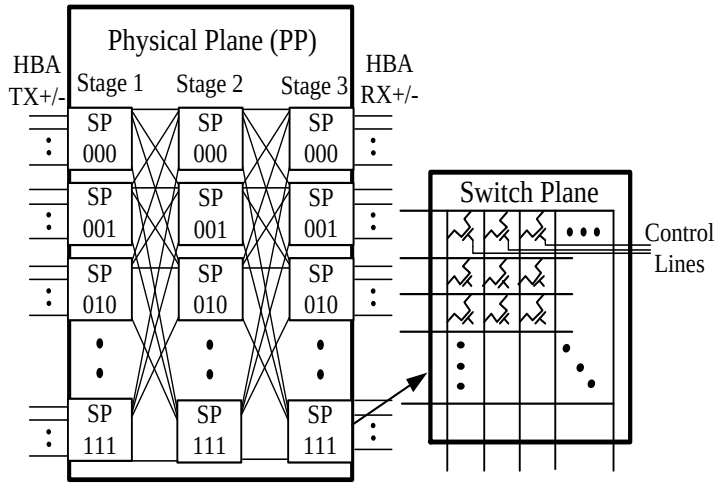


Figure 2. The structure of Physical Plane

V. Conclusion and Applicatons

Our research addresses the need for a byte addressed memory area network (MeMAN). This cloud memory architecture promises to remove the barrier that is currently limiting virtualized computing CPUs from scaling and having proximity beyond the memory bus architecture. This research addresses the various cloud memory architectures and explores the hypotheses involved, the scale or scope for each architecture, as well as the associated limitations (delay, throughput, and maximum size).

Applications for MeMAN are diverse. Our current research focuses on a primary application of cloud memory whereby the memory associated with a user's personal virtual data storage is extended to a personal private device [15]. The academic community, such as the large educational network of King AbdulAziz University, is one subject of research of how a new concept of personal cloud memory interacts with the larger public and private clouds.

Virtual desktops (VDT) are virtual machines for which personal data, personal applications, and hardware and software resources (CPU, OS, memory, and data storage) are provided and shared in the cloud. The concept is analogous to virtual servers which are essentially files that can be instantiated on physical servers as needed. Virtual server migration is facilitated by hypervisors that are network based operating systems on top of computer OS and hardware. VDT, similar to desktop and notebook computers, carries the personal data, software, and computing resources within the network, accessed through thin clients connecting the VDT users to the cloud.

While the replacement of desktop (DT) with VDT in the cloud has numerous advantages, such as data, hardware, and apps sharing; backup and reliability assurance; as well as reduced operational cost, VDT is often constrained by security issues as well as network and bandwidth bottlenecks. The combination of a personal cloud and a public cloud provides the benefits of personal computing and cloud computing, namely a social context while maintaining privacy and security through local and encrypted data storage [18].

As an example, students can carry their personal data and virtual machines loaded with their personal applications on a thumb drive[15], plug the thumb drive into thin clients that may perform local computation, and when needed, access the public cloud for social computing and data synchronization. Our research involves defining the level of privacy, security, and applications needed by students for the application of personal cloud memory. Given the user requirements, cloud memory architectures are being evaluated for implementing a personal memory cloud.

The research is aimed at social networking as well, aiming first to look at how a personal memory cloud enhances the educational and social experience in a large university such as KAU, as well as the cost reduction possible without massive investment by students on computers and data storage, made worse with the lack of data backup, software administration [16], and disaster recovery. While this research is practical in nature, the research impact can be significant for providing a broader cloud memory experience at KAU.

References

[1] Joseph Y Hui, David A Daniel, "Virtualization of a Host Computer's Native I/O System Architecture," U.S. Patent 7 734 859, 6/8/2010.

[2] Joseph Y. Hui, David A Daniel, Tim Jeffries, "Memory Area Networks for Extended Computer Systems," U.S. Patent 12 589 448(pending), 10/23/2009.

[3] Joseph Y Hui, David A Daniel, "Time-space Carrier Sense Multiple Access," U.S. Patent 6 141 355, 7/19/2011.

[4] Joseph Y Hui, David A Daniel, "Control Architecture and Implementation for Switches with Carrier Sensing," U.S. Patent 12/655,096 (granted), 11/22/2011.

[5] Joseph Y. Hui, David A. Daniel, "Terabit Ethernet: Access and Core Switching Using Time-Space Carrier Senseing," *IEEE Systems Journal*, vol. 4 issue 4, pp. 458-455, 2010.

[6] Nikolaos Hardavellas, Galen C. Hunt, Sotiris Ioannidis, Robert Stets, Sandhya Dwarkadas, Leonidas Kontothanassis, and Michael L. Scott (1997). *Efficient Use of Memory-Mapped Network Interfaces for Shared Memory Computing*. Available: http://tab.computer.org/tcca/NEWS/mar97 /sandhya.pdf

[7] Hongzhou Zhao (2007). *Shard Memory Computing on Networks of Workstations*. Available: http://www.cs.rochester.edu/u/sandhya/csc258/.../zhao_treadmarks.pdf

[8] H. Su, B. Gordon, S. Oral, A. George. *SCI Networking for Shared-Memory Computing in UPC: Blueprints of the GASNet SCI Conduit*. Available: http://gasnet.cs.berkeley.edu/SuGordon-HSLN.pdf

[9] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir (1983). *The NYU Ultracomputer-Designing an MIMD*. Available: http://www.cs.utexas.edu/~dburger/teaching/cs395t-s08/papers/7_ultracomputer_a.pdf

[10] *Shared Memory Parallel Computer*. Available: http://www.cs.utexas.edu/~dburger/cs395t/papers/ 7_ultracomputer_b.pdf

[11] National Center for Ecological Analysis and Synthesis (2011). *High-Performance Raster/Grid Computing: Multiple CPUs and Shared Memory*. Available: http://www.nceas.ucsb.edu/scicomp/usecases/ HpcProgWithMpiSharedMemory

[12] David A. Badera, Guojing Cong (2006). *Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs*. Available: http://www.cc.gatech.edu/~bader/papers /MST-JPDC.pdf

[13] Henry Neeman, Charlie Peck (2011). *Parallel Programming & Cluster Computing Shared Memory Multithreading*. Available: http://symposium2011.oscer.ou.edu/.../oksupercompsymp2011 _sipe_share...

[14] Mark Hill. *Distributed Shared Memory on IP Networks*. Available: http://www.research.google.com/pubs/archive/49.pdf

[15] Premium USB Blog (2011). *Why the USB Flash Drive and Cloud Storage can Coexist*. Available: http://blog.premiumusb.com/2011/05/why-the-usb-flash-drive-and-cloud-storage-can-coexist/

[16] VMware Community (2011). *ESXi Installation on a USB Pen Drive*. Available: http://communities.vmware.com /message/1869154

[17] Isaac Agudo, David Nuñez, Gabriele Giammatteo, Panagiotis Rizomiliotis, Costas Lambrinoudakis. *Cryptography goes to the Cloud*. Available: http://www.nics.uma.es/sites /default/files/Crypto_STAVE.pdf

[18] Yanpei Chen, Vern Paxson, Randy H. Katz (2010). *What's New About Cloud Computing Security?* Avaiable: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-5.pdf

[19] Richard Chow, Philippe Golle, Markus Jakobsson, Ryusuke Masuoka, Jesus Molina, Elaine Shi, Jessica Staddon (2009). *Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control*. Available: http://www.eecs.berkeley.edu/~elaines/docs /ccsw.pdf

[20] Jon Oberheide, Kaushik Veeraraghavan, Evan Cooke, Jason Flinn, Farnam Jahanian. *Virtualized In-Cloud Security Services for Mobile Devices*. Available: http://www.eecs.umich.edu/fjgroup /pubs/mobivirt08-mobilecloud.pdf.

[21] Traian Andrei (2009). *Cloud Computing Challenges and Related Security Issues*. Available: http://www.cs.wustl.edu/~jain/cse571-09/ftp/cloud/index.html

[22] David A. Daniel, "Design, Modeling, Simulation, Circuit Design, and Proof-of-Concept for Transporting PCI Data Over Ethernet," Ph.D. thesis, Dept. Elect. Eng., Arizona State University, AZ, 2010.

[23] MindShare, Inc., Tom Shanley, Don Anderson, *PCI System Architecture*, Fourth Edition, Addision Wesley, 1999.

[24] PCI-SIG, *PCI Express Base Specification Revision 2.0*, 2007.

[25] "Low-Voltage Differential Signaling," Design Note, Texas Instruments, 2000.

[26] Joseph Y. Hui and David A. Daniel, "Terabit Ethernet: a Time-Space Carrier Sense Multiple Access Method," *IEEE Globecom* 2008.

[27] Apurva N Sahasrabudhe, "Terabit Ethernet," M.S. thesis, Dept. Elect. Eng., Arizona State University, AZ, 2009.

Real-Time Communication Protocol with Temporally Enhanced Erasure Codes

Seonho Choi[1], Hyeonsang Eom[2]
[1]Department of Computer Science, Bowie State University, Bowie, Maryland, U.S.A.
[2]School of Computer Science and Engineering, Seoul National University, Seoul, Korea

Abstract— *In Forward Error Correction (FEC) technique, redundant encoded packets are transmitted to and decoded by receiver(s) so that up to a certain number of lost packets can be recovered by using those redundant packets. The way the encoded packets are generated is to, first, divide a stream of data packets into blocks of k packets, and to generate n encoded repair packets for each block using the original packets. The receiver can recover the k original packets as long as it receives at least k distinct packets in each block. Hence, the encoding/decoding in such protocols are performed on block-by-block basis.*

In [20], a new technique, named as Temporally Enhanced Erasure Codes (TEEC), was introduced as an enhancement to the existing erasure coding technique. In TEEC the scopes of encoding/decoding are expanded beyond block boundaries, and they may overlap with scopes of neighboring blocks. In this paper, we apply the TEEC technique in designing a real-time communication protocol whose objective is to increase the packet recovery rate while reducing packet recovery delays as much as possible at the receiver side.

In terms of average packet loss rates and end-to-end delays for packet delivery, it is shown that a protocol employing TEEC outperforms block-by-block based protocols.

Keywords — reliable protocol, error recovery, FEC, erasure codes, protocol, real-time

1 Introduction

A wide range of applications in the future Internet will require both reliable and timely transmission of data packets between a sender and receiver(s). These include network whiteboards and video/audio conferencing systems, etc. Providing both reliability and timeliness in packet transmission is a difficult problem and several approaches have been proposed and used.

Two different mechanisms exist for error control in network applications, Automatic Repeat reQuest (ARQ) and Forward Error Correction (FEC). ARQ technique is based on timeout and retransmission of packets, and has been adopted in TCP protocol.

However, ARQ technique has its limitations especially if there exist timing constraints on packet delivery times as in real-time multimedia applications, or if it is used for multicasting packets to many receivers. FEC can be used in those environments in which ARQ cannot be used effectively [12,10,7,6]. The sender sends additional packets along with the original data packets so that, in cases of packet losses, they can be used to recover the original data packets that were lost during transmission. One advantage of this approach is that no further interaction between the sender and receiver is needed as long as the lost packets can be recovered from the received packets. In FEC, the sender performs encoding and the receiver performs decoding to reconstruct the lost data packets [10,7,18]. One of the well-known codes in FEC is erasure codes [10]. In erasure codes, the encoded packets are generated by, first, dividing a stream of data packets into blocks of k packets, and generating n-k (>0) encoded repair packets for each block using the original packets. The sender sends n packets for each block, and the receiver can recover the k original data packets as long as it receives at least k distinct original or repair packets. Hence, the encoding and decoding in such protocols are performed on block-by-block basis. FEC technique has been used in many applications due to the advantages mentioned above. For example, the applications and performances of FEC were studied for multicast applications, and it was shown that FEC can be a viable solution approach for reliable multicast transmissions [6, 7, 12, 15].

A new technique named as *Temporally Enhanced Forward Error Correction (TEFEC)* was proposed as an enhancement to the existing block-based FEC codes such as erasure and Tornado codes, and it was applied to design a reliable communication protocol [20]. In this paper, we apply the TEEC technique in designing a real-time communication protocol whose objective is to increase the packet recovery rate while reducing packet recovery delays as much as possible at the receiver side.

The basic idea of TEFEC is that the scopes of encoding and decoding may be expanded beyond block boundaries, and they may overlap with scopes of neighboring blocks. The encoding scope factor is defined to be the number of blocks from which the encoded packets will be generated. For example, Figure 1 shows how the encoding is performed in TEFEC with the encoding scope factor 2.

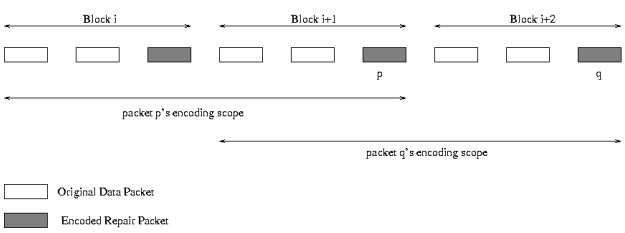


Figure 1 TEFEC with encoding scope factor of 2

In this paper, the detailed algorithm of TEEC is presented with their performances compared to those of the block-based erasure codes. TEEC was developed and implemented by extending erasure codes [10, 20]. To show their applicability, a real-time protocol is developed and simulated by using TEEC technique. It is shown that, in most of the cases, the error correcting capability could be enhanced much while not reducing the average recovery delays at the receiver. The expense we have to pay for this is slightly increased decoding overhead at the receiver. TEEC can be utilized in many applications to enhance the error correcting capability without increasing the bandwidth used for encoded packets and without affecting the real-time aspects of the applications. If the network path has long delays, or if it is expensive to send back NAK packets due to power problems as in mobile communication devices, TEEC can also be used to enhance the performances. For multicast applications, FEC was shown to be a viable solution approach [6,7,12,15]. TEEC can be effectively utilized in such applications as a better solution approach due to the enhanced error-correcting capabilities.

The paper is organized as follows. Section 2 summarizes the erasure codes [10,4]. In Section 3 the encoding and decoding algorithms are presented for TEEC, and the detailed real-time protocol is presented in Section 4 which utilizes TEEC. In Section 5 the simulation settings and results are presented for the reliable protocol with TEEC and traditional protocols with block-based FEC techniques. Finally, in Section 6 the conclusion of the paper follows.

2 Erasure Codes

A brief introduction to the erasure code is given in this section with its principles and computational complexities. A more detailed description of the codes can be found in numerous literatures [1,4,9,10]. The erasure code presented and used in this paper is called *linear block codes* whose principles and implementation techniques are nicely presented in [10].

In erasure codes the original data packets to be sent by the sender are divided into independent blocks whose size (in terms of the number of packets) is denoted by *k*. Using *k* data packets, *n (>k) encoded* packets are generated and transmitted instead of *k* data packets. Thus, the sender transmits total n packets for each block. One important property of erasure codes is that the receiver can *decode* and recover all *k* data packets as long as it successfully receives *k* distinct packets. In other words, up to *n-k* packet losses can be recovered without requiring retransmission of packets. This type of code is called *(n,k)* code.

If the packet size becomes large, then it may become difficult to apply encoding/decoding to entire packets. However, as is shown in [10] large packets can be split into multiple data items, like bytes, and the encoding/decoding process can be efficiently applied by taking one data item per packet. Under this assumption, a block is redefined to be a collection of data items from different packets instead as a collection of data packets. Hence, from a collection of k data packets, σ blocks will be created each of which contains *k* data items where σ is a packet size in terms of data items.

2.1 Linear Block Code

Let $\boldsymbol{x}= x_0\ x_1\ \ldots\ x_{k-1}$ be the source data consisting of k data items. Then an *(n,k)* linear code can be obtained by

$$\mathbf{y} = G\mathbf{x}$$

where *G* is an $n \times k$ generator matrix satisfying some properties that will be explained later, and $\boldsymbol{y}$ is an $n \times 1$ matrix containing *n* data items which will be sent by the sender. This explains how the encoding is performed.

Let $\boldsymbol{y}$ be denoted as $y_0\ y_1\ \ldots\ y_{n-1}$, and suppose that the receiver gets any subset of *k* data items in $\boldsymbol{y}$, and let $i_0\ i_1\ \ldots\ i_{k-1}$ denote the indices corresponding to the data items successfully arrived at the receiver. Then, a $k \times k$ matrix can be obtained from *G* by selecting the rows of *G* whose row indices match $i_0\ i_1\ \ldots\ i_{k-1}$. This new matrix is denoted as *G'* and it is known that any *k* rows of *G* are linearly independent, which is a necessary property for obtaining original data items in $\boldsymbol{x}$ from $\boldsymbol{y}(i_0)\ \boldsymbol{y}(i_1)\ \ldots\ \boldsymbol{y}(i_{k-1})$ by solving

$$(G')^{-1}\ \mathbf{y} = \mathbf{x}$$

where $\boldsymbol{y}(i_h)$ denotes a data item in $\boldsymbol{y}$ whose index is equal to i_h. Hence, the decoding process involves finding an invert matrix of *G'* and multiplying it to $\boldsymbol{y}$. For the purpose of later usages, let $\boldsymbol{x}^m$ and $\boldsymbol{y}^m$ denote vectors of original and encoded data items for a block *m* where $m \geq 0$.

By applying simple algebraic operations we can transform *G* so that upper *k* rows are equal to $k \times k$ identity matrix. This means that the first *k* data items in $\boldsymbol{y}$ are equal to *k* original data items in $\boldsymbol{x}$, and this type of code is called a *systematic code* [10]. If a systematic code is used to encode blocks, the decoding process may be simplified in case the number of lost packets small. In this paper, it is assumed that the systematic code is used and applied.

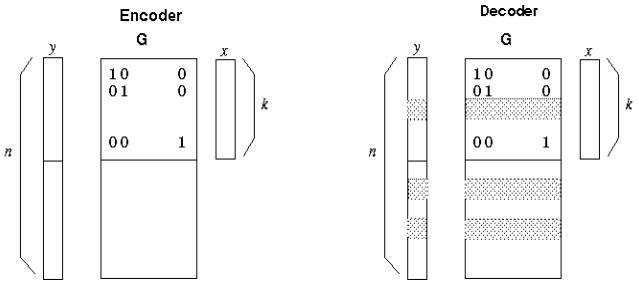


Figure 2. Matrix representation of encoding and decoding using systematic code.

However, there exist some limitations of erasure (or linear block) code. One of them is the precision overhead that incurs during matrix multiplications. That is, consider the matrix multiplication $\boldsymbol{y} = G\boldsymbol{x}$. If the number of bits used for representing one data item in $\boldsymbol{x}$ is α bits, then representing $\boldsymbol{y}$'s data items may require more data bits since they have to be found by multiplying the matrix rows to $\boldsymbol{x}$. This may result in

significant overhead [10], and may invalidate the usage of erasure code despite of its nice loss-recovery capabilities.

This problem can be avoided if the encoding and decoding is performed in a finite field instead of real number field. The algebraic operations are performed in an extension field, $GF(p^r)$, where *GF* stands for Galois Field. The addition and multiplication operations in $GF(p^r)$ can be efficiently performed using XOR operations.

The matrix known as Vandermonde matrix can be used as a generator matrix for encoding and decoding in $GF(p^r)$. The coefficients of the matrix is of the form:

$$g_{ij} = x_i^{j-1}$$

where the x_i's are elements of $GF(p^r)$. If all x_i's are different, the matrix is invertible. Another useful property is that, if no x_i is equal to 0, then the matrix can be transformed into a simpler form which has an identity submatrix in the first *k* upper rows in the matrix. This enables us to build a systematic code.

2.2 Complexity of Encoding/Decoding

When *n-k* repair packets are generated by the sender, the complexity of encoding process is $O((n-k)k\sigma)$ where σ denotes the number of data items in a packet. The total cost of decoding a block is $O(kl^2+kl\sigma)$ where *l* denotes the number of lost packets in the block. Note that $l \le min(k, n-k)$ should hold. These results are presented in more detail in [10].

3 Temporally Enhanced Erasure Codes

The erasure code techniques explained in the previous section usually perform encoding and decoding using blocks as their basic units. The blocks are mutually independent in a sense that encoding and decoding of one block don't affect those for other blocks. The common approach is to divide the entire data packet stream into packet blocks and apply encoding and decoding algorithms to each block.

A new approach, Temporally Enhanced Erasure Codes (TEEC), has been developed to reduce the amount of bandwidths required for additional repair packets, and to reduce the end-to-end delays for packet deliveries. In other words TEEC tries to enhance the repair capacity when the same amount of bandwidth is used for repair packets as in traditional erasure codes while not increasing the recovery delays at the receiver. In this section the conditions and processes for encoding/decoding blocks in TEEC are explained under the assumption that the sender sends *n* encoded (*k* original + *n-k* repair) packets for each block and the receiver tries to recover lost packets if it is feasible.

3.1 Encoding in TEEC

The key idea of TEEC is that the repair packets for a block *m* are calculated from data items that belong to *(v-1)* previous blocks, *m-(v-1), m-(v-2), …, m-1,* as well as from those in the block *m. v (≥1)* is called an *encoding scope factor*. This is represented by the following equation.

$$\mathbf{y}^m = G_{n\times vk}\,[\mathbf{x}^{m-(v-1)}, \mathbf{x}^{m-(v-2)}, \ldots, \mathbf{x}^m]^T \quad (1)$$

where $G_{n\times vk}$ is a generator matrix of size $n\times vk$, and $[\boldsymbol{x}^{m-(v-1)}, \boldsymbol{x}^{m-(v-2)}, \ldots, \boldsymbol{x}^m]^T$ is a $vk\times 1$ matrix that consist of *vk* data items in *v* blocks including block *m*. The generator matrix $G_{n\times vk}$ is obtained from the original generator matrix G^0 which can be found from a Vandermonde matrix in a Galois Field, $GF(p^r)$, as in [10]. Let the size of G^0 be equal to $n_0 \times vk$ where $n_0 \ge n$. Note that the upper *vk* rows of G^0 is equal to an identity matrix. Then, the algorithm is given in the following:

Algorithm 1

Insert an identity matrix of size $k \times k$ as a submatrix into upper right corner of $G_{n\times vk}$, and fill zeros into the remaining part of the *k* upper rows.

Let $r_1, r_2, \ldots, r_{n-k}$ denote *n-k* row vectors[1] chosen from G^0 whose row indices are greater than or equal to *vk*. Also, let $r_i(x)$ and $r_j(y)$ denote the *x*-th and *y*-th elements in the vectors r_i and r_j, respectively, where $1 \le x < y \le vk$. In addition, let *l* be defined as $\lfloor (vk-y)/k \rfloor$. Then, the *n-k* row vectors chosen above can be inserted as the row vectors in $G_{n\times vk}$ with new row indices, $k, k+1, \ldots, n$, if the following relationship is satisfied:

$$\forall\ i,j,t,x,y\ :\ 1 \le i,j \le n\text{-}k,\ \ 0 \le t \le l,\ \ 1 \le x < y \le vk\ ::\ r_i(x)\cdot r_j(y+tk) - r_i(y)\cdot r_j(x+tk) \ne 0\ \vee\ (i=j \wedge l=0)$$

The second step in Algorithm 1 guarantees that, if any two elements (in *x*-th and *y*-th columns) are chosen in an *i*-th row in $G_{n\times vk}$ where $i \ge k$, and if we choose any two other elements whose column indices are *tk* apart from *x* and *y*, respectively, then a 2×2 matrix obtained from these four elements has a rank of 2. If $i=j$ and $l=0$, then this condition is not enforced.

To find out maximum number of rows in G^0 satisfying the above condition, we developed a heuristic algorithm which incrementally builds the set of rows (from G^0) that satisfy the above condition. The Galois Field $GF(2^8)$ was used to generate the original generator matrices. It was shown in [20] that, for large encoding scope factors (*v*) and block sizes, no repair rows may be found. But, for encoding scope factors up to 5 or 6 and for reasonably large block sizes around 10, the valid repair rows could still be found while the more valid rows can be obtained for smaller encoding scope factors and block sizes.

3.2 Decoding in TEEC

The decoding process is more complicated than the encoding process. One important characteristic of the decoding process is that a variable number of consecutive blocks are grouped together and decoding is performed for the lost packets in them. Once the decoding is successfully finished, then all the lost packets are recovered. The minimum number of consecutive blocks that need to be grouped together for decoding is *v* as is seen from the equation (1). For example, the lost packets in the block *m* may be recovered immediately after the packets for *m* arrive when all the data packets for the

[1] The chosen rows don't need to be consecutively located in G^0.

previous *v-1* blocks are available, or after the packets of future blocks arrive at the receiver. A *decoding scope factor*, *w (≥v)*, is defined as the maximum number of blocks which can be grouped together for decoding. The decoding can be performed using less number of blocks than *w* if certain conditions are satisfied, which will be derived in this section.

Suppose that the packets for blocks, *z, z+1, z+2,..., z+c,* are available at the receiver where $0 \le c \le w-1$, and the receiver wants to perform decoding for those blocks. The following assumptions are made regarding the decoding process:

The total number of packets used as input to the decoding process is equal to $(c+1)k$.

Only the repair packets that belong to the blocks $z+v-1, z+v, \ldots, z+c$, are used in decoding. This is because the repair packets that belong to the blocks, $z, z+1, \ldots, z+v-2$, were encoded using the original data packets that belong to the blocks prior to *z*.

If the total number of received packets is greater than $(c+1)k$, then it is assumed that $(c+1)k$ packets are chosen among them[2] and the rest of the packets are discarded. Following notations are defined once the $(c+1)k$ packets are selected:

λ_j denotes the number of received original packets (among *k* packets sent by the sender) for block *j*.

δ_j denotes the number of received repair packets (among *n-k* repair packets initially sent) for block *j*.

Then the following equation should hold:

$$\sum_{i=z+v-1}^{z+c} \delta_i + \sum_{i=z}^{z+c} \lambda_i = (c+1)k$$

As a second step of the decoding process, the matrix of size $(c+1)k \times (c+1)k$ is obtained by Algorithm 2 and is denoted as $G'_{(c+1)k,(c+1)k}$.

Algorithm 2

Assign indices to $(c+1)k$ packets in the range of 1 to $(c+1)k$. Repeat the following steps 2 through 4 for each packet whose index is *p* where $1 \le p \le (c+1)k$.

Let *j* denote a block index in $[0,c]$ and *i* denote a row index in $[1,n]$ for the *p*-th packet. That is, the *p*-th packet belongs to the block $z+j$, and the packet was encoded by using the *i*-th row vector in $G_{n \times vk}$.

From $G_{n \times vk}$ extract a row corresponding to the index *i*. Let this row vector be denoted as r_i.

(i) if the *p*-th packet is a data packet,

→ If $0 \le j \le v-1$, then discard the first $(v-j-1)k$ number of 0 elements from r_i and append *0*s to the end of r_i to make it of size $(c+1)k$

→ Otherwise, add $(j-v+1)k$ number of *0*s to r_i and append *0*s to the end of r_i to make it of size $(c+1)k$.

(ii) if the *p*-th packet is a repair packet,

→ add $(j-v+1)k$ number of *0*s to r_i and append *0*s to the end of r_i to make it of size $(c+1)k$.

4. Place the vector obtained in step 3 as a *p*-th row in $G'_{(c+1)k,(c+1)k}$.

If the obtained matrix, $G'_{(c+1)k,(c+1)k}$, is invertible, then the following equation can be utilized to recover the lost packets in the blocks $z, z+1, z+2, \ldots, z+c$.

$$\mathbf{y}=G'_{(b+v)k,(b+v)k}\ [\mathbf{x}^z,\mathbf{x}^{z+1},\mathbf{x}^{z+2},\ldots,\mathbf{x}^{z+c}]^T$$

where some items in the vectors $x^z, x^{z+1},\ldots,x^{z+c}$ may not be known and should be recovered by the decoding algorithm. The above equation can be rewritten as follows:

$$(G'_{(b+v)k,(b+v)k})^{-1}\mathbf{y}= [x^z,x^{z+1}, x^{z+2},\ldots,x^{z+c}]^T$$

The lost packets in the right hand side may be found by multiplying the inverse matrix to the vector $\boldsymbol{y}$ which contains the arrived packets.

Theorem 1

The matrix, $G'_{(c+1)k,(c+1)k}$, obtained by Algorithm 2 has an inverse matrix only if the following conditions are satisfied:

$$\forall j : 0 \le j \le v-1 ::$$
$$\sum_{i=z+v-1}^{\min(z+v-1+j,z+c)} \delta_i + \sum_{i=z}^{z+j} \lambda_i - (j+1)k \ge 0 \qquad (1)$$

$$\forall j : v \le j \le c ::$$
$$\sum_{i=j}^{\min(j+v-1,c)} \delta_{z+i} + \lambda_{z+j} - k + \min\left(0, \sum_{i=z+v-1}^{z+j-1} \delta_i + \sum_{i=z}^{z+j-1} \lambda_i - jk\right) \ge 0 \qquad (2)$$

$$\forall j : 0 \le j \le c ::$$
$$\sum_{i=z+v-1}^{z+j} \delta_i + \sum_{i=z}^{z+j} \lambda_i \le (j+1)k \qquad (3)$$

Proof: The proof may be found in [20].

The meaning of this theorem is that the three conditions should hold when the obtained matrix, $G'_{(c+1)k,(c+1)k}$, is invertible. That is, those three conditions are necessary conditions for decodability, and it is possible that $G'_{(c+1)k,(c+1)k}$ is not invertible even if the three conditions hold. However, it has been shown from the simulations that the probability of such occurrences is very low when the encoding is performed using $G_{n \times vk}$ obtained by Algorithm 2. This means that the three conditions in Theorem 1 are tight and may be used to detect the invertibility of $G'_{(c+1)k,(c+1)k}$ in most of the cases.

In [20], the detailed explanation of these conditions are given along with how they may be checked in real implementation.

4 Real-Time Protocol Using TEEC

In this section, a real-time protocol is presented which utilizes TEEC encoding/decoding algorithms to recover lost packets during transmission. No retransmission of lost packets are performed. This assumption makes sense especially in real-time applications which has certain timing constraints. It is also assumed that the real-time data packets are generated periodically at the sender and sent to the receiver. The original

[2] How these packets should be chosen will be explained later after the main theorem is presented regarding the decodability conditions.

data packet stream is divided into blocks that contain *k* packets, and the sender also sends *n-k* redundant (encoded) repair packets along with the *k* data packets in each block. These *n* packets are sent to the receiver in a block period.

Suppose that the packets for blocks, *z, z+1, …, z+c*, have arrived at the receiver where $0 \le c \le w-1$, and let $z+i$ denote an index of the first block that has at least one lost data packet where $0 \le i \le v-1$. Again, note that the decoding scope factor, *w*, denotes a maximum number of consecutive blocks that can be grouped together for decoding. The following algorithm is run whenever the transmission of the packets for each block is completed at the receiver[3]. Let $z+c$ denote the block index whose packet transmission is completed now.

Three variables, *SBI*, *EBI*, and *FLBI*, are defined and used, which represent the start block index, the end block index of a decoding scope, and the first loss block index within a decoding scope, respectively. Given *c+1* blocks described above, *SBI* and *EBI* are initialized to *z* and *z+c*. *FLBI* denotes the index of the first block that has at least one lost data packet between *SBI* and *EBI*. This variable is set to $z+i$.

Algorithm 3

Input: *SBI*, *EBI*, and the packets for the blocks in *[SBI,EBI]*
Output: Recovered packets

1. Find out *FLBI* value in *[SBI,EBI]*.
2. If no *FLBI* is found, then the following variables are reset, and exit:
   - *SBI = SBI+1*
   - *EBI = EBI+1*
3. Check condition (3) of Theorem 1 for the last block: find
$$\sum_{l=SBI+v-1}^{EBI} \delta_l + \sum_{l=SBI}^{EBI} \lambda_l - (EBI - SBI + 1)k$$
If the above term is positive, then discard that number of repair packets in the block, *EBI*, since they violate the condition (3) of Theorem 1.
4. If $EBI-SBI+1 \le w$ holds, and (1) ∧ (2) of Theorem 1 holds,
   - Obtain a decoding matrix for the blocks. If the decoding matrix has an inverse matrix, then the lost packets in the blocks are recovered. Recovery of the lost packets are noted. Then, the following variables are reset for the next invocation of the algorithm, and the current invocation is exited:
     - *SBI = EBI-v+2*
     - *EBI = EBI+1*
   - If the decoding matrix is not invertible, then the receiver gives up recovering lost packets in the block, *FLBI*.
     - if $FLBI+1 \le EBI$, the algorithm is repeated for the blocks *FLBI+1, FLBI+2, …, EBI* by performing the following steps:
       - *SBI = FLBI+1*
       - Goto step 1.
     - If *FLBI=EBI*, then set *SBI=EBI+1*, *EBI=EBI+1*, and exit.
5. If $EBI-SBI+1<w$ holds, and the condition (1) ∧ (2) of Theorem 1 doesn't hold,
   - if there is a possibility that (1) ∧ (2) of Theorem 1 will hold after receiving future block packets,
     - In this case, set *EBI=EBI+1*, and exit from the algorithm.
   - if there is no possibility that (1) ∧ (2) of Theorem 1 holds even after receiving future block packets,
     - if $FLBI+1 \le EBI$, then set *SBI=FLBI+1* and goto step 1.
     - If *FLBI=EBI*, then set *SBI=EBI+1* and *EBI=EBI+1*, and exit.
6. If $EBI-SBI+1=w$ holds, and (1) ∧ (2) of Theorem 1 doesn't hold, then, the following steps are performed:
   - if $FLBI+1 \le EBI$, then set *SBI=FLBI+1* and goto step 1.
   - If *FLBI=EBI*, then set *SBI=EBI+1* and *EBI=EBI+1*, and exit.

As can be seen from the above procedure, the receiver may utilize *v-1* previous blocks that are already recovered when decoding the following blocks. This is the ideal case in which the benefits of TEEC can be fully exploited. If, for some reason, the group of blocks can't be decoded, then decoding of the first block that has some lost packets will be given up, and the receiver will proceed the decoding process starting from the next block.

In the step 5 above, it is needed to decide whether it is worthwhile to wait for future block packets or not. That can be decided by examining the following formula.

$$\mu_{EBI} = \max\left(-\begin{pmatrix}\delta_{EBI} + (w - EBI + SBI - 1)(n-k) + \lambda_{EBI} - k + \\ \min(0, \sum_{i=SBI}^{EBI-1}\delta_i + \sum_{i=SBI}^{EBI-1}\lambda_i - (EBI - SBI)k)\end{pmatrix}, 0\right)$$

This formula is obtained from the condition (1) and (2) of Theorem 1. If μ_{EBI} is greater than 0, then it means that the blocks can't be decoded even if the future block packets arrive successfully. If it is equal to 0, then it means that it is worthwhile to wait for more future block packets since there exists a possibility.

5 Simulation

The simulation program was written to test the performance of the reliable protocol utilizing TEEC, and the results are compared with those utilizing the traditional erasure codes. The sender continuously sends the real-time data packets with a period, *P*. Three parameter values are used by the sender for encoding, *n*, *k*, and *v* (for TEEC). Also, for decoding, the receiver has a parameter, *w*. With a set of given parameters, three different cases were simulated:

[3] This time can be detected by observing the packets belonging to the future block arrives at the receiver, or by a timeout.

1. Traditional erasure codes with block size n (k original plus n-k redundant packets).
2. TEEC with parameters, n, k, v, w.
3. Traditional erasure codes with block size vn (vk originals plus $v(n-k)$ redundant packets).

In all three cases, it is assumed that the packets are sent with the same inter-packet departure times. Especially, the time period during which n packets are sent by the sender is denoted as P.

By comparing the performances of the first and second schemes, we can identify how much we can benefit by using the TEEC based protocol instead of using traditional block-based FEC schemes.

Note that the third method uses a large value for block size with the same bandwidth fraction used for sending repair packets as in the first and second methods, and is considered here to see what the advantage is by using TEEC compared to just increasing the block size with the same bandwidth fraction used for repair packets.

The loss models we assumed in this simulation are the independent loss model and the burst loss model which is obtained by a two state Markov chain. The independent loss model has one parameter, p, denting the probability that a packet will be lost during transmission. The Markov chain has two parameters, p_1, p_2, for the transition probability from normal to normal state and the transition probability from loss state to loss sate, respectively.

The metrics obtained and compared in this simulation are as follows:

- Average packet loss rate.
- Average packet recovery delay, i.e., the amount of time taken for the receiver to wait until the lost packet is reconstructed. The delays are measured in units of P, i.e., multiples of P.

The simulation was run for 20000 blocks for each set of parameter values. Figures 3 through 6 were obtained with parameters of $n=8$, $k=5$, $v=5$, and $w=15$, for independent packet loss model and burst packet loss model. Figure 3 shows that the average packet loss rates for TEEC is lower than those for cases 2 and 3 while Figure 4 shows that the average packet recovery delays for TEEC is much lower than those for case 3 and slightly higher than those for case 2. These two figures show the effectiveness of TEEC compared to traditional erasure codes. For burst packet loss model, Figures 5 and 6 show the results with p_1 set to 0.9. Figure 5 shows that, for most of the values of p_2, TEEC has the lowest average packet loss rates and Figure 6 shows that the average packet recovery delays lie between those of case 1 and case 2 up to $p_2=0.65$. It increases after that point and becomes largest for larger p_2 values. Larger p_2 value means longer burst length. Similar results were obtained for different sets of parameter values and models as in this case.

6 Conclusions

A real-time communication protocol utilizing TEEC have been developed in this paper, and its effectiveness was shown through the simulations. The simulation results of the protocol show that TEEC enhances the performances of the protocol in terms of end-to-end delays and average packet loss rates. Even though the experiments were conducted on unicast communication models, more benefits from TEEC can be obtained in multicast applications since many receivers may take advantages of TEEC.

7 Acknowledgement

This work was supported in part by US Army Research Office (ARO) grant W911NF-12-1-0060, and in part by Seoul R&BD Program (No. JP110034). The ICT at Seoul National University provided research facilities for this study.

8 References

[1] R. E. Blahut, "Theory and Practice of Error Control Codes", Addison Wesley, MA, 1984

[2] S. McCanne, V. Jacobson, and M. Vetterli, "Receiver-driven Layered Multicast", ACM SIGCOMM'96, August 1996, Stanford, CA, pp.1-14.

[3] A. McAuley, "Reliable Broadband Communication Using a Burst Erasure Correcting Code", Proc. SIGCOMM'90.

[4] S. Lin, D. J. Costello, "Error Control Coding: Fundamentals and Applications", Prentice Hall, 1983.

[5] S. Lin, D. J. Costello, M. Miller, "Automatic-repeat-request error-control schemes", IEEE Comm. Magazine, v.22, n.12, pp.5-17, Dec. 1984.

[6] J. Nonnenmacher, E.W. Biersack, "Reliable Multicast: Where to use Forward Error Correction", Proc. 5th Workshop on Protocols for High Speed Networks, pp.134-148, Sophia Antipolis, France, Oct. 1996.

[7] J. Nonnenmacher, E.W. Biersack, D. Towsley, "Parity-Based Loss Recovery for Reliable Multicast Transmission", IEEE/ACM Transactions on Networking, 6(4):349-361, August 1998.

[8] R. G. Kermode, "Scoped Hybrid Automatic Repeat reQuest with Forward Error Correction", Proc. SIGCOMM'98.

[9] V. Pless, "Introduction to Error-Correcting Codes", 2nd edition, Wiley, 1989.

[10] L. Rizzo, "Effective Erasure Codes for Reliable Computer Communication Protocols", ACM Computer Communication Review, Vol.27, n.2, Apr. 1997, pp. 24-36.

[11] L. Rizzo, Sources for an erasure code based on Vandermonde matrices. Available at http://www.iet.unipi.it/~luigi/vdm.tgz

[12] L. Rizzo, L. Vicisano, "A Reliable Multicast data Distribution Protocol based on software FEC techniques", DEIT Technical Report LR-970116.

[13] Y. Wang, S. Lin, "A modified selective-repeat type-II hybrid ARQ system and its performance analysis", IEEE Transactions on Communication, v.COM-31, n.5, pp.593-608, May 1983.

[14] J. Lin, S. Paul, "RMTP: A Reliable Multicast Transport Protocol", IEEE INFOCOM'96, March 1996, pp.1414-1424.
[15] D. Rubenstein, J. Kurose, D. Towsley, "Real-Time Reliable Multicast Using Proactive Forward Error Correction", Technical Report 98-19, Dept. of Computer Science, Univ. of Mass., Amherst, March 1998.
[16] K. Perumalla, A. Ogielski, R. Fujimoto, "MetaTeD: A Meta Language for Modeling Telecommunication Networks", GIT-CC-96-32, Technical Report, College of Computing, Georgia Institute of Technology, 1997.
[17] Seonho Choi,"Temporally Enhanced Erasure Codes for Reliable Communication Protocols", available upon request to seonho@cs.bowiestate.edu. Dept. of Computer Science, Bowie State University, Bowie, MD 20715.
[18] M. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. Spielman, and V. Stemann. *Practical loss-resilient codes*. In Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pages 150--159, 1997.
[19] Schacham, N., and McKenny, P., "Packet Recovery in High-Speed Networks using Coding", In Proceedings of IEEE INFOCOMM'90.
[20] Seonho Choi, "Temporally Enhanced Erasure Codes for Reliable Communication Protocols", Computer Networks Journal, Vol. 38, p.713-730, 2002.

Appendix

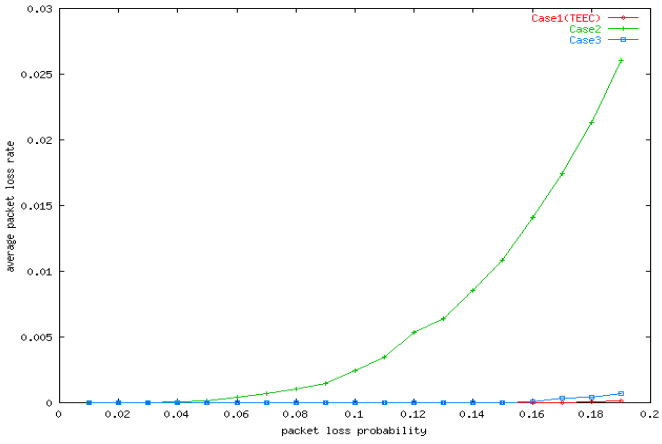


Figure 3. Average packet loss rate for n=8, k=5, v=5, w=15, and independent packet loss model.

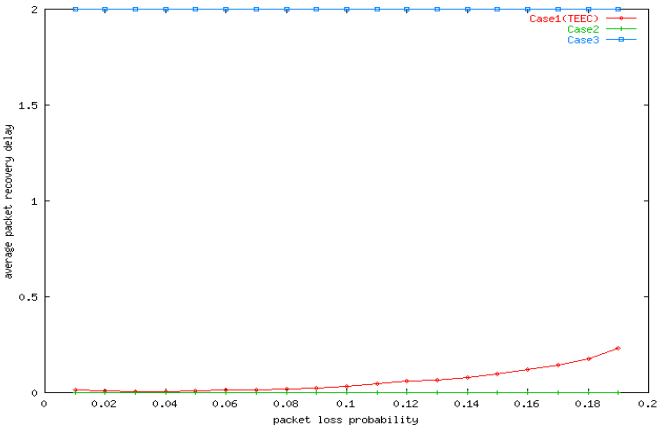


Figure 4. Average packet recovery delay for n=8, k=5, v=5, w=15, and independent packet loss model.

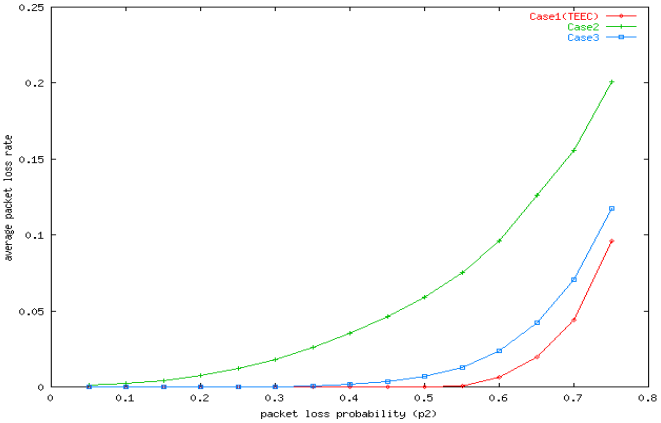


Figure 5. Average packet loss rate for n=8, k=5, v=5, w=15, and burst loss model with p_1=0.9.

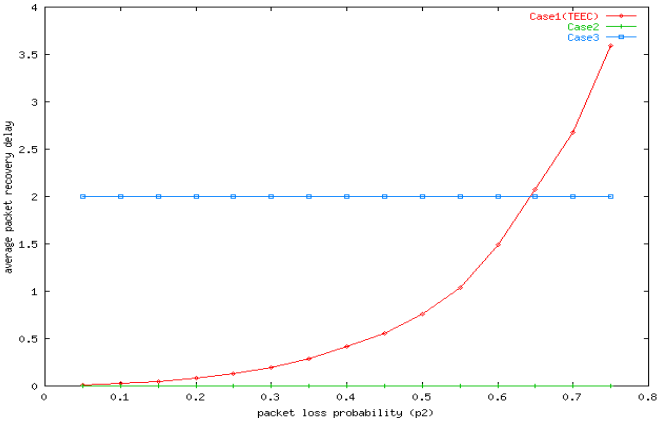


Figure 6. Average packet recovery delay for n=8, k=5, v=5, w=15, and burst loss model with p_1=0.9.

Predictive Model for FFT Scalability Performance

K.M. Mostafa[#1], M. B. Abdelhalim[#2], Mohamed Waleed Fakhr[#3]

[#]*College of Computing and Information Technology (CCIT)*
Arab Academy of Science and Technology and Maritime Transport (AASTMT)
Cairo, Egypt
[1]kziad@live.com
[2]mbakr@ieee.org
[3]waleedf@aast.edu

Abstract— In order to introduce more efficient high performance computing (HPC) applications we need to construct a performance model for the complex heterogeneous systems.
Most of the HPC applications, as in the molecular dynamics simulation implement the FFT parallel algorithm that consumes a major portion of the application execution time.
This work constructs a model that can be used to decide the execution plan scalability and highlight major factors that impact directly the algorithm performance.
The parallel FFT algorithm is used as a case study for our modelling procedure.
This paper aims to explore the most impacting parameters of the parallel FFT algorithms execution where other platform and hardware-dependent factors are not included.

Keywords: FFT, Performance, model, parallel, decomposition

1 Introduction

High Performance Computing (HPC) allows scientists and engineers to solve complex science and engineering problems using parallel algorithms on a large number of computing processors and high bandwidth network.

As the scalability is a very important measurement of the applications performance, predicting the algorithm scalability is required in the pre-deploying phase.

The peta-scale computers are the new generation of the HPC platforms as in blue waters project. It delivers 1 peta folps performance [1], the lack of scalability of the parallel algorithms forms a step challenging issue in the new Peta-scale computers.

The scalability issues may prevent the applications from achieving the desired performance hence rise up the importance of the prediction model.

The performance prediction model should address the major factors that affect the performance, however in order to implement a generic model it should not be very detailed to a specific hardware or a specific platform.

The Fast Fourier Transform (FFT) is an enhanced method for calculating the Discrete Fourier Transform (DFT). As it is more efficient, often reducing the computation steps by hundreds. The 3D FFT can be expressed as [2]

$$f(k_x,k_y,k_z)=\sum_z\left[\sum_y\left[\sum_x f(x,y,z)\cdot e^{ik_x x}\right]e^{ik_y y}\right]e^{ik_z z} \quad (1)$$

FFT has been one of the most popular and widely used numerical methods in many areas of scientific computing, including digital speech and signal processing, solving partial differential equations, molecular dynamics many-body simulations and Monte Carlo simulations.

Given its importance, there have been a large number of libraries that provide different implementations of FFT aimed at achieving high-performance in various environments.

The outline of this paper is organized as follows. Section 2 shows the most relevant work related to our work. Section 3 describes the different decomposition algorithms .The experiments test-bed and the analytical model described in section 4 and 5 respectively , section 6 illustrate the experiments results and finally paper conclusion is section 7.

2 Related Work

The steps required to calculate 3DFFT with 2D decomposition add extra computation and communication step while they extend the algorithm scalability [3].

Previous work introduced an analysis of the 3DFFT decompositions methodologies and the performance measurements of each decomposition method [3], the experimental results shown in Table 1 shows a speedup of the 2D decomposition [4], the 2D decomposition theoretical analysis proves a better scalability performance up to the product of the widest two dimensions of the data mesh however it is proven in this work that other factors have a major impact on the algorithm scalability.
The major FFT parallel algorithm cost introduced as a summation of the communication and computation tasks Tfft(N; P;Chunks) = Tcomp(N; P) + Tcomm(Chunks; P)

Where Tcomp(N; P) is the Computation Time, Tcomm(Chunks; P) is the Communication Time [5] . In this paper the analysis model cost equation introduced with more details.

Many intensive studies addressed the FFT algorithm communication cost highlighting BlueGene/L torus and mesh network topologies [6], in our work both computation and communication factors considered along with the different decomposition algorithms and the grid size, the model addressed only the major network parameters aiming to be independent of the network topologies.

A comparison between different FFT libraries and the performance measurements through each decomposition method of the libraries predicts the theoretical scalability enhancements of the 2D decomposition [4], The FFTW [7] is a popular implementation of the 3DFFT; however the FFTW does not support the 2D decomposition however some modifications added to enable the 2D decomposition more details in section IV .

Message Passing Interface (MPI) is a language-independent communications protocol used to program parallel computers especially with the distributed system. MPI communication protocols include both point-to-point and collective communication [8].

There are various models to evaluate the MPI communication performance from the hardware perspective as LogP [9] , LogGP [10] and pLogP [11] where parameters as in Table 2 [12] .

TACC ranger network parameters measured [12] for the blocking and the non-blocking communication (discussed in the analytical model section) as shown in Table 3.

Table 1:256^3 1D & 2D decomposition results

Library	Time[1]	Cores[2]	Cores/Node	Decomposition	Nodes
2DECOMP&FFT	0.0045	8192	16	2D	512
2DECOMP&FFT	0.0057	512	1	2D	512
P3DFFT	0.006	4096	16	2D	256
P3DFFT	0.0063	512	1	2D	512
FFTW	0.0124	256+	1	1D	256
P3DFFT	0.013	256	1	1D	256
2DECOMP&FFT	0.014	256	1	1D	256
P3DFFT	0.0233	256	16	1D	16
2DECOMP&FFT	0.0244	256	16	1D	16
FFTW	0.0341	256	16	1D	16

1 Libraries sorted by time.
2 Total number of cores
+ Maximum number of total cores (1D decomposition).

Table 2: LogGP Parameters

Parameter	Description
m	Message size
L	Network latency
o	Overhead per message transmission
g	Gap between successive message transmissions
G	The reciprocal of network bandwidth
P	Number of processors

Table 3: Network latency values

Parameter	Blocking	Non-Blocking
L	1.95 µs	1.75 µs
o	0.65 µs	0.85 µs
g	0.65 µs	0.85 µs

3 3D FFT decompositions

FFT Decomposition is the process of distributing the 3D FFT data mesh and determine the steps of communication and computation across the number of processors grid.

The decomposition way beside the hardware architecture and specifications are the key factors of any FFT algorithm execution plan.

In 1D decomposition shown in Figure **1** which is also called slab decomposition, the 3D data grid is divided into a number of slabs across one dimension

At the first step, the 1DFFT first calculated with respect to two dimensions let's say the x and y direction, and along the third direction z then in the second step a global transpose with all to all communication takes place then the at the last step the 2DFFT is calculated across the last dimension z

In the 1D decomposition, the global transpose (all-to-all communication) takes place only once, however the drawback of this algorithm is the scalability where the number of slabs is limited to the maximum number of processors in one dimension.

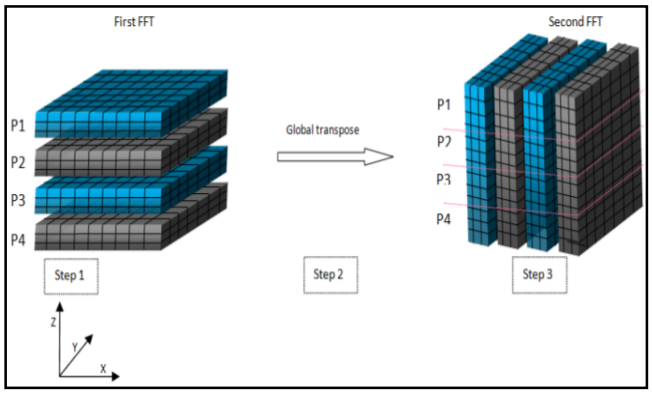


Figure 1 : 3DFFT calculation steps for the 2D decomposition

2D decomposition of the data, which sometimes called the pencil decomposition, is when the data is divided across two dimensions of the 3D data grid. In the first step, the 1DFFT is calculated for each pencil followed by a global transpose in the second step. At the third step, the 1DFFT is calculated along the second dimension then again in the forth step the global transpose takes place. At the last step, a final calculation of the 1DFFT along the last dimension is performed.

The 2D decomposition algorithm is shown in Figure 2 [6]. Two global all-to-all transpose takes place, however

the maximum scalability is up two the product of the widest two dimensions of the 3D data grid.

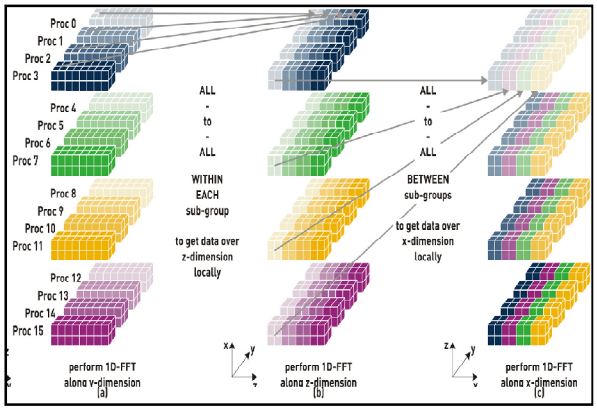


Figure 2: 3DFFT calculation steps for the 2D decomposition

4 Test-bed

Our target platform is the Texas Advanced Computing Centre – Ranger constellation –TACC Ranger. The TACC system consists of 62,976 nodes interconnected through InfiniBand technology providing a theoretical 1GB/sec point-to-point bandwidth, TACC provide a peak performance of 9.2 GFLOPS/core or 128 GFLOPS/node [13]. .

As shown in Figure **3**, every four cores are attached to one socket. Such structure must be considered when measuring the communication performance specially when testing with little number of cores (e.g. 2, 4, 8, and 16) to avoid measuring the intra socket communication instead of the network communication.

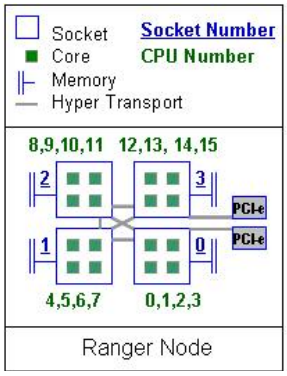


Figure 3: TACC Ranger Architecture

Our experiments code was developed using C++, MPI routines version 1.2.7, PGI 7.1 compiler.

The Fastest Fourier Transform in the West (FFTW) [7] library is selected for computing multiple dimensions of the DFT, FFTW is a MIT developed C subroutine library for computing multiple dimensions of the DFT, FFTW supports only the 1D decomposition however some changes are added to our experimental code to enable the 2D decomposition testing.

In the 1D decomposition, the ***fftw_mpi_local_size*** function used to determine the local size for each processor, ***fftw_plan_dft*** function is used to initialize the plan with the local variables and the input and output data and ***fftw_execute*** function is used to perform the FFT calculation.

In the 1D decomposition, the local size of the processors that are out of boundaries (greater than the largest axis) is equal to zero. In our implementation of the 2D decomposition we overwrite the local size variables to scale it to the number of the product of the widest two dimensions.

In the transposing task, the ***fftw_mpi_plan_transpose*** function used to transpose the global 3D grid over the contributors processors, in the 2D decomposition the transposing action is modified to divide the contributing processors to a set of sub groups that share the required transposing axis as shown in Figure 2. Similar related work in [11] implement the same idea.

Performance Application Programming Interface (PAPI) [14] framework supports several events API, PAPI 3.6.0 is used to measure the number of the floating points instructions needed to calculate the FFT for each processor, however PAPI does not support MPI intercommunication profiling.

Integrated Profiling Monitoring IPM [15] is a portable profiling infrastructure for parallel codes. It provides performance profile from several aspects of the computation communication, and IO of the parallel programs, IPM used to measure the communication time acquired, the message size transferred, and the number of messages for the transposing tasks.

5 The analytical model

This paper aims to explore the most impacting parameters of the parallel FFT algorithms execution to build a high level model for its timing behaviour. The model takes into consideration the major hardware system parameters. However it is not hardware-specific as the more generic parameters we achieve, the more platforms we can use.

For simplicity we can divide the FFT algorithm into two non overlapping stages.

First: the computation part where each processor computes the DFT for the local portion of the data grid, the number of steps per core based on the algorithm order time (N^2, N log N, etc) and data mesh size.

As shown in equation (2) the FFTW algorithm uses O(N log N) [7], the computation time per core depends on several parameters.

The $\boldsymbol{FFC}$ parameter is the floating point factor that reflects the number of flops required for calculating each point.

$\boldsymbol{FFC}$ equals 5 in case of complex number transform as in our experiments and 2.5 in case of real number transform [7], $\boldsymbol{f(Nprocs)}$ is the maximum number of processors scalability considering the FFT algorithm decomposition method where $\boldsymbol{f(Nprocs)}$ in 1D decomposition equals the widest dimension of the data grid. In the 2D dimension, it is equal to the product of the widest two dimensions of the data grid as shown in

equation (3). **Peak Performance** is the peak floating-point operations performance per core, the value of the peak performance equal to 9.2 GFLOPS/core or 128 GFLOPS/node [13]. **Peak Percentage** is the average peak performance percentage measured in our experiments and is found to be around 950 mflops that equivalent to 10% of the theoretical peak performance as shown in Figure 4, the **mflops** is the rate of flops × 10^6 per second where **flipns** stands for the number of floating points measured, **Ndata** is the number of the data grid points.

.

$$\boldsymbol{computation\ time} = \frac{FFC \times \left(\frac{N_{data}}{f(Nprocs)}\right) \log_2 \frac{N_{data}}{f(Nprocs)}}{Peak\ performance \times Peakpercentage} \quad (2)$$

$$\boldsymbol{f(Nprocs)} = \begin{cases} \boldsymbol{Max(N_X, N_Y, N_Z)}, & 1D \\ \boldsymbol{N_i * N_j\ AND\ N_i, N_J \neq Min(N_X, N_Y, N_Z)}, & 2D \end{cases} \quad (3)$$

Second: The communication part that represents the time consumed in the transposing tasks of the algorithm.

Again, the transposing time can be divided into two parts, first the data transmission time, as we have first to figure out the data grid size that will be transposed by each core by dividing the total data size $\boldsymbol{N_{data}}$ on the number of processors $\boldsymbol{f(Nprocs)}$ that depends directly on the decomposition way multiplied by the number of bytes in each data grid point **FFM**, divided on the network bandwidth **BW**. The second part is the overall transpose Message latency $\boldsymbol{MSG_{lat}}$ that equal to the summation of the three overhead parameters multiplied by the number of messages $\boldsymbol{f(Nprocs)}$ as in the transposing task, an all-to-all communication takes place.

The overhead parameters derived from three parameters l, o, and g. l refers to network latency, o overhead per message transmission and g gap between the successive message transmissions. The three parameters have been measured in previous work [12] as in Table **2** and Table 3, the tables address the latency values for the MPI blocking and non-blocking communication.

The blocking MPI means that the program execution will be suspended till the message buffer is safe to use, however in the non-blocking MPI communication, the call returns immediately after the call is initiated. The FFTW is found to use the non blocking MPI_SENDRECV routine in the transposing communications.

It is worthy to mention that the communication performance has many other parameters that are more hardware-specific; it is out of the scope of this model as the model is designed to be algorithm-oriented.

$$\boldsymbol{communication\ time} = \boldsymbol{MSG_{lat}} + \frac{\left(\frac{N_{data} * FFM}{f(Nprocs)}\right)}{\boldsymbol{BW}} \quad (4)$$

$$\boldsymbol{MSG_{lat}} = \boldsymbol{(L + o + g) * f(Nprocs) - 1} \quad (5)$$

The number of points handled by each core can be calculated as $N_{data}/\ f(Nprocs)$ multiplied by the number of bytes FFM.

6 Model Verification

This section discusses and compares the model and the algorithm performance and scalability.

In the following Figures, the performance results measured in seconds over 2 to 4096 cores, multiple runs executed to calculate the average execution time for 64^3 and 256^3 input grids, respectively.

As discussed in the previous Section, the model and the actual computation time are measured by calculating the number of flops of the FFT calculations for each core divided by the peak performance multiplied by %10 as the peak percentage.

The *MPI_Sendrecv* mpi routine used in point-to-point communication by the FFTW algorithm, the *MPI_Allreduce* used to gather the results of the average number of flops through the running instances, the communication analysis include only the *MPI_Sendrecv* communication in the transposing tasks excluding any other communication overhead used in gathering or synchronizing the data.

The IPM used to profile the messages between the cores, as simple histogram method used to calculate the average message time between cores.

In our platform, TACC Ranger; some parameters should be addressed to submit a job, one of them is the wayness [13] as it determines the number of cores to be used through the whole node (16 cores), for example if the program will run on 32 core and 8 wayness method used, this means that the program will need 4 nodes and then only 8 cores will be used in each node.

As shown in Figure **3** each group of 4 cores are communicated with the other groups through one socket, In the small number of processors (Nprocs < 16), the 1-way job scheduling wayness used to avoid measuring the inter-group communication instead of the InfiniBand socket communication.

6.1 1-D Decomposition

The model successfully predicts the overall trend changes in the FFT execution performance and scalability, Figure 5 and Figure 6 show that the time consumed by the computation part is dominant when the number of cores is less than 64, the communication and computation parts start to be very close when the number of cores are equal to the widest dimension over 2, the results confirms that algorithm scalability extent is equal to the widest dimension in the input 3D grid, when the number of cores is larger than the widest dimension the algorithm divides the data grid on a number of cores equals to the widest dimension and the rest of the cores is not involved in the execution plan.

6.2 2-D Decomposition

The model successfully predicts the overall performance behaviour and scalability, as shown in Figure 7 and Figure 8, the 2-D decomposition success to scale to the product of the widest two dimensions of the data grid as discussed in the previous Section, however the performance drops dramatically when the number of all-to-all messages increased to be more than 128×128 and

the size of the message became very small (less than 32 bytes) where the message latency time is the dominant. The message sizes transmitted for each scenario are shown in Figure 9 that shows the relationship between the message size of each scenario and the decomposition algorithm as well as the data grid size, message size is measured in bytes, the message size of the all-to-all communication step can be derived from the communication analytical equation as $\frac{N_{data}*FFM}{f(Nprocs)^2}$ where the total amount of data required to be transposed on each processor is divided over the total number of the processors engaged in the all-to-all communication.

As shown in equation (5) the number of messages directly impact the time consumed by the messages latency, as shown in Figure 9 the dramatic increase of the messages number along the increase of the number of processors.

As shown in previous work [12] and in Figure 12 the relation between the message size and the G parameter - that introduced in Table 2 - has a Monotonic decrease relationship where the bandwidth affected badly by the message size.

To achieve the desired scalability and performance, the message size and the network latency factors should be considered.

In Figure 10 and Figure 11 the model accuracy for the gap between the predicted time and the measured time is shown for both 1-D and 2-D decompositions with 64^3 and 256^3 data grids inputs, the gap fluctuations may refer to change in the peak performance due to hardware considerations however the overall performance behaviour still predictable.

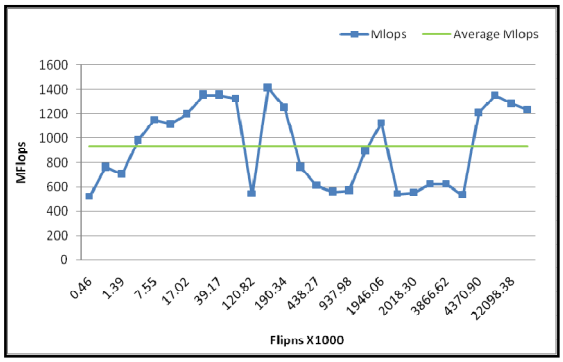


Figure 4: Average Number of mflops

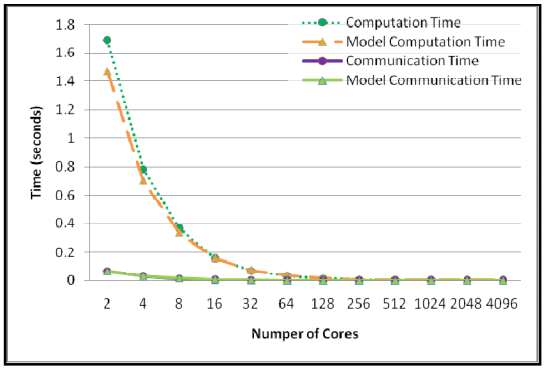


Figure 5: 1D decomposition 256^3 data grid size

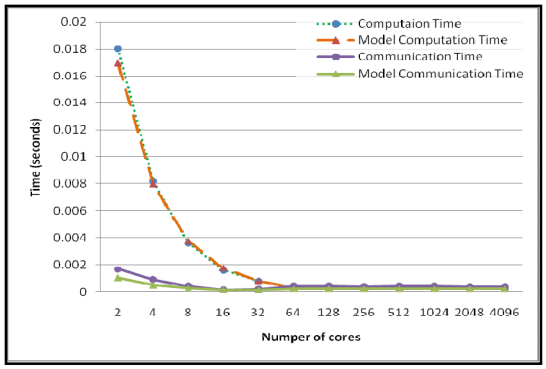


Figure 6: 1D decomposition 64^3 data grid size

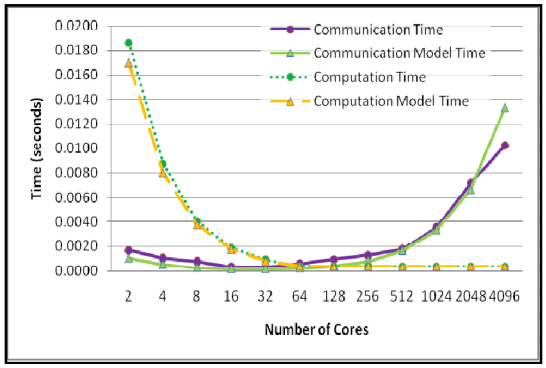


Figure 7: 2D decomposition 64^3 data grid size

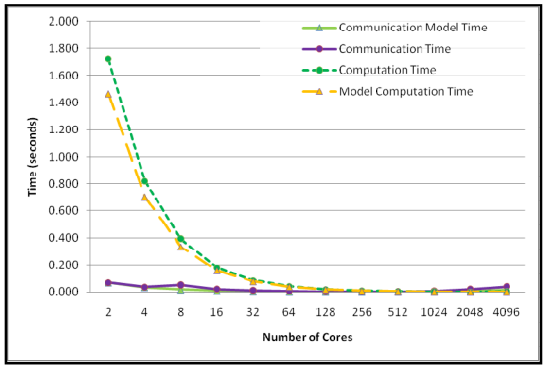


Figure 8:2D decomposition 256^3 data grid size

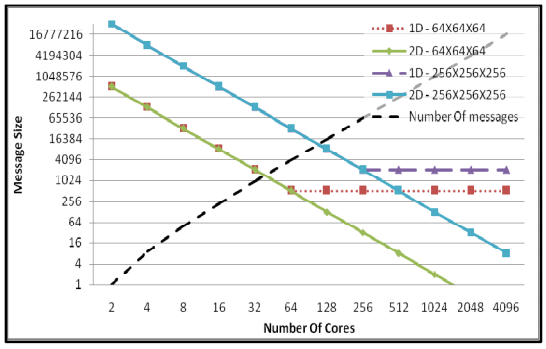


Figure 9 : Message size and number of messages

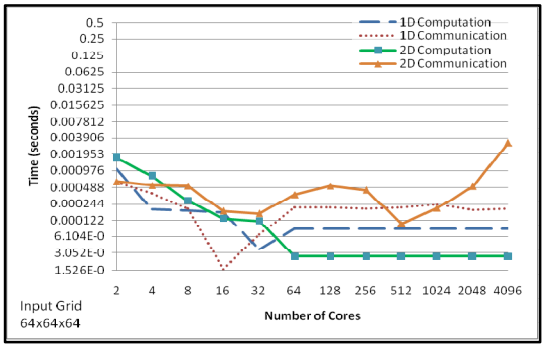


Figure 10: Gap between the Model and the Measured for 64^3

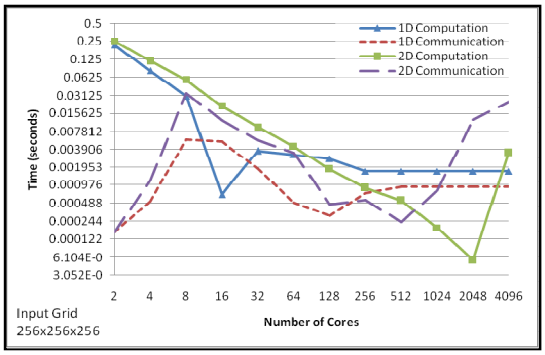


Figure 11: Gap between the Model and the Measured for 256^3

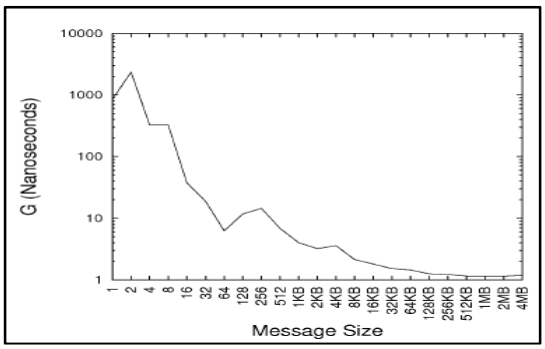


Figure 12: G for Various Message Sizes

7 Conclusions and future work

The paper present a predictive model of scalability and performance of the parallel FFT algorithm, The model successfully predicts the overall scalability performance behavior, although there is a gap between the predicted and measured performance for the very large and the very small message size.

The model scope includes the execution of 3DFFT grid, 1D and 2D decomposition methods. A future work can be done against the 3D decomposition (volumetric decomposition).

The model can be enhanced to include the computation and communication overlapping probability.

The network parameters, as the bandwidth and the message latencies parameters should be known for the target platform; it can be done through 1- System documentation 2- profiling tools.

8 Acknowledgment

Our deep thanks go to Eric J. Bohm - Department of Computer Science - University of Illinois at Urbana-Champaign for his support and for providing HPC resources that have contributed to the research results reported within this paper.

9 References

[1] National Center for Supercomputing Applications. [Online]. http://www.ncsa.illinois.edu/BlueWaters, accessed at May 2012.

[2] P. Balaji, W. Gropp, R. Thakur, and A. Chan, "Communication Analysis of Parallel 3D FFT for Flat Cartesian Meshes on Large Blue Gene Systems," in *HiPC'08 Proceedings of the 15th international conference on High performance computing*, Springer-Verlag.

[3] Roland Schulz, "3D FFT with 2D decomposition," 2008.

[4] Evangelos Brachos, "Parallel FFT Libraries," University of Edinburgh, MSc in High Performance Computing 2011.

[5] Manisha Gajbe et al., "Auto-Tuning Distributed-Memory 3-Dimensional Fast Fourier Transforms on the Cray XT4," in *Proc.~Cray User's Group (CUG) Meeting*, Atlanta, 2009.

[6] H. Jagode, "Fourier Transforms for the BlueGene/L Communications Network," University of Edinburgh, Master's thesis 2006.

[7] Fastest Fourier Transform in the West. [Online]. http://www.fftw.org/

[8] Message Passing Interface. [Online]. http://www.mpi-forum.org/

[9] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. J, "pLogP :Incorporating long messages into the LogP model for parallel computation," *Journal of Parallel and Distributed*, pp. 71-79, Aug 1997.

[10] D. E. Culler et al., "LogP: Towards a realistic model of parallel computation.," in *In Proc. of the 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, 1993, pp. 1-12.

[11] T. Kielmann, H. E. Bal, and K. Verstoep, "Fast measurement of LogP parameters for message passing platforms," in *14th IEEE Intl, Works. on Parallel and Distributed Processing (IPDPS)*, 2000, pp. 1176–1183.

[12] Verdi March, Vijayaraghavan Murali, Yong Meng Teo, Simon See, and James T. Himer, "Towards Predictive Modeling of Message-Passing Communication," in *IEEE International Conference on High Performance Computing and Communications*, 2009, p. 6.

[13] Texas Advanced Computer Center. [Online]. http://www.tacc.utexas.edu/user-services/user-guides/ranger-user-guide accessed at April 2012

[14] Performance Application Programming Interface. [Online]. http://icl.cs.utk.edu/papi/

[15] Integrated profiling monitoring. [Online]. http://ipm-hpc.sourceforge.net/ accessed at April 2012

Apriori-Map/Reduce Algorithm

Jongwook Woo
Computer Information Systems Department
California State University
Los Angeles, CA

Abstract –*Map/Reduce algorithm has received highlights as cloud computing services with Hadoop frameworks were provided. Thus, there have been many approaches to convert many sequential algorithms to the corresponding Map/Reduce algorithms. The paper presents Map/Reduce algorithm of the legacy Apriori algorithm that has been popular to collect the item sets frequently occurred in order to compose Association Rule in Data Mining. Theoretically, it shows that the proposed algorithm provides high performance computing depending on the number of Map and Reduce nodes.*

Keywords: Map/Reduce, apriori algorithm, Data Mining, Association Rule, Hadoop, Cloud Computing

1 Introduction

People started looking at and implementing Map/Reduce algorithm for most of applications, especially for computing Big Data that are greater than peta-bytes as cloud computing services are provided, for example, by Amazon AWS.

Big Data has been generated in the areas of business application such as smart phone and social networking applications. Especially these days, the better computing power is more necessary in the area of Data Mining, which analyzes tera- or peta-bytes of data. Thus, the paper presents *Apriori-Map/Reduce Algorithm* that implements and executes *Apriori algorithm* on Map/Reduce framework.

In this paper, section 2 is related work. Section 3 describes the legacy *apriori algorithm*. Section 4 introduces Map/Reduce and Hadoop and presents the proposed *Apriori-Map/Reduce algorithm*. Section 5 is conclusion.

2 Related Work

Association Rule or Affinity Analysis is the fundamental data mining analysis to find the co-occurrence relationships like purchase behavior of customers. The analysis is legacy in sequential computation so that many data mining books never resist illustrating it.

Aster Data has SQL MapReduce framework as a product [9]. Aster provides *nPath SQL* to process big data stored in the DB. Market Basket Analysis is executed on the framework but it is based on its SQL API with MapReduce Database.

Woo et al [11-13] presents Market Basket Analysis algorithms with Map/Reduce, which proposes the algorithm with *(key, value)* pair and execute the code on Map/Reduce platform. However, it does not use the *apriori property* but instead adopts *joining* function to produce paired items, which possibly computes unnecessary data.

3 Apriori Algorithm

Apriori algorithm shown in Figure 3.1 has been used to generate the frequent item sets in the amount of data transactions in order to produce an association rule.

```
Map transaction t in data source to all Map nodes;
//(1)
C1 = {size 1 frequent items};
// (2) min_support = num / total items; for example: 33%
L1 = {size 1 frequent items ⋂ min_support};


for (k = 1; Lk !=∅; k++) do begin


    // (3) sort to remove duplicated items
    Ck+1 = Lk join_sort Lk;

    for each transaction t in data source with Ck+1 do
        // (4)
        increment the count of all candidates in Ck+1 that
        are contained in t
        // (5) find Lk+1 with Ck+1 and min_support
        Lk+1 = {size k+1 frequent items ⋂
          min_support};
    end
end


return ∪k Lk;
```

Figure 3.1. Apriori Algorithm

The association rule has been used efficiently to manage stock items and products etc analyzing the customer's behavior. It is based on Apriority Property where all subsets of a frequent item set must also be frequent.

For example, minimum support is 2 and there are size 2 item sets generated: *<[coffee, cracker], 3>* and *<[coke,*

cracker], 1> as *<[item pairs], frequency>*. And, when there are size 3 item sets produced as *<[coffee, cracker, milk]>* and *<[coke, cracker, milk]>,* as *Apriority Property*, *<[coke, cracker, milk]>* is eliminated before counting the frequencies of the item sets in the transaction data, which reduce unnecessary computing time.

The time complexity of the algorithm is $O(k \times (k^2 + t \times n))$ when k: size of frequent items, t: transaction data, n: number of item elements in each transaction t. It is simplified to $O(k^3 + k \times t \times n)$ and then $O(k \times t \times n)$ where $t >> k$, $n>>k$.

4 Apriori-Map/Reduce Algorithm

4.1 Map/Reduce in Hadoop

Map/Reduce is an algorithm used in Artificial Intelligence as functional programming. It has been received the highlight since re-introduced by Google to solve the problems to analyze Big Data, defined as more than peta-bytes of data in distributed computing environment. It is composed of two functions to specify, "Map" and "Reduce". They are both defined to process data structured in *(key, value)* pairs.

Inspired by Google's *MapReduce* and *GFS* (Google File Systems) [1], Map/Reduce platform is implemented as *Apache Hadoop* project that develops open-source software for reliable, scalable, and distributed computing. Hadoop can compose hundreds of nodes that process and compute Big Data. Hadoop has been used by a global community of contributors such as Yahoo, Facebook, Cloudera, and Twitters etc. Hadoop provides many subprojects including *Hadoop Common*, *HDFS*, *MapReduce*, *Avro*, *Chukwa*, *HBase*, *Hive*, *Mahout*, *Pig*, and *ZooKeeper* etc [2].

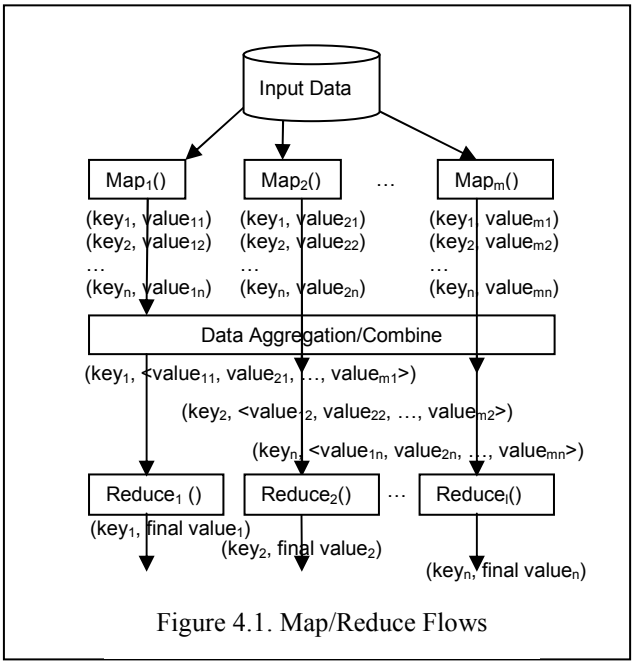


Figure 4.1. Map/Reduce Flows

The map and reduce functions run on distributed nodes in parallel. Each map and reduce operation can be processed independently on each node and all the operations can be performed in parallel. Map/Reduce can handle Big Data sets as data are distributed on *HDFS* (Hadoop Distributed File Systems) and operations move close to data for better performance [5].

Hadoop is restricted or partial parallel programming platform because it needs to collect data of *(key, value)* pairs as input and parallely computes and generates the list of *(key, value)* as output. In map function, the master node divides the input into smaller sub-problems, and distributes those to worker nodes.

Map transaction *t* in data source to all Map nodes;
//(1) In each Map node *m*
C_{m1} = {size 1 frequent items at the node *m*};
// (2) In Reduce, compute C_1 and L_1 with all C_{m1} ;
C_1 = {size 1 frequent items};
// (3) min_support = num / total items; for example: 33%
L_1 = {size 1 frequent items ∩ min_support};

for ($k = 1$; L_k !=∅; k++) do begin

// (4) In each Map node *m*
// L_{mk}: L_k mapped to each node m;
// sort to remove duplicated items
$C_{m(k+1)} = L_k$ *join_sort* L_{mk};

// (5) In Reduce, use Apriori Property
compute C_{k+1} with all sorted $C_{m(k+1)}$;
if (k>=3) prune(C_{k+1});

for each transaction *t* in data source with C_{k+1} do
// (6) In each Map node *m*
increment the count of all candidates in $L_{m(k+1)}$ that are contained in *t*
end
// (7) In Reduce, find L_{k+1} with $L_{m(k+1)}$ and
// min_support
L_{k+1} = {size *k+1* frequent items ∩ min_support};
end

return ∪k L_k;

Figure 4.2. Apriori-Map/Reduce Algorithm

Figure 4.1 illustrates Map/Reduce control flow where, as m is map node id, each valuemn is simply 1 and gets accumulated for the occurrence of items together, which is clearly illustrated in Market Basket Analysis Algorithm of Woo et al [11-13]. Map function takes inputs (k_1, v_1) and generates $<k_2, v_2>$ where < > represents list or set. Combiner function that resides on map node takes inputs $(k_2, <v_2>)$ and generates $<k_2, v_2>$. Reduce function takes inputs $(k_2, <v_2>)$ and generates $<k_3, v_3>$.

Map/Reduce approach, especially for big data set, gives us the opportunity to develop new systems and evolve IT in parallel computing environment. The approach started a few years ago but many IT companies in the world already have adapted to Map/Reduce approach.

4.2 Apriori-Map/Reduce Algorithm

Figure 4.2 is the proposed *Apriori-Map/Reduce Algorithm* that runs on parallel Map/Reduce framework such as Apache Hadoop. *prune*(C_{k+1}) function is to remove the non-frequent item set C_{k+1} by eliminating non-frequent item sets C_k as non-frequent item sets cannot be a subset of frequent item sets.

The algorithm starts with (1) that calculates frequent item set for each map node as the time complexity $O(t/m \times n)$ when t: number of transactions, n: number of items in the transactions, m: num of map nodes. Then, (2) are to collect the frequent item set and (3) is to remove items that does not meet the minimum support in reduce nodes as $O(t/r \times n)$ when r: number of reduce nodes. The time complexity of (1-3) can be simplified initially to $O((t/m + t/r) \times n)$ and then $O(t \times n / x)$ when $m = r = p$.

(4) is to calculate frequent item set with an additional item by joining, sorting, and eliminating the duplicated items in each map node where *join_sort* is $O(k \times k / m)$ when k: size of frequent items and *prune* is $O((k-1) \times k / r)$ that is simplified to $O(k^2/r)$. Similarly, (5) is to collect the frequent item set at the reduce nodes. (6) is to count item frequencies that do not meet the minimum support at the map nodes as $O(t/m \times n)$. (7) is to remove items that does not meet the minimum support in reduce nodes as $O(t/r \times n)$ that is initially simplified to $O((t/m + t/r) \times n)$ and then $O(t \times n / p)$ when $m = r = p$.

4.2.1 Time Complexity

The overall time complexity of *Apriori-Map/Reduce Algorithm* is calculated as $O(k \times (k2 + t \times n)/p)$ where k: size of frequent items, t: number of transactions, n: number of items in the transactions, p: number of map and reduce nodes assuming the node sizes are the same. And, it becomes $O((k3 + k \times t \times n)/p)$ and then $O(k \times t \times n/p)$ where $t >> k$, $n >> k$. It theoretically shows that the time complexity is p times less than the sequential *apriori algorithm*.

4.2.2 Example of the Algorithm

Figure 4.3 is the example transaction data at a store to explain how the proposed algorithm works.

Transaction 1: cracker, beer
Transaction 2: chicken, pizza,
Transaction 3: coke, cracker, beer
Transaction 4: coke, cracker
Transaction 5: beer, chicken, coke
Transaction 6: chicken, coke

Figure 4.3 Transaction data at a store

Suppose that minimum support is 2/6 that represents two items out of 6 transactions as 33%.

(a) The first step
Assuming there are three Map nodes, two transaction data are distributed to three map nodes, that is, map node 1 has transaction data 1 and 2. Node 2 has transaction data 3 and 4. Node 3 has transaction data 5 and 6. Therefore, we can generate size 1 frequent items C_{m1} with an item pair set <*item, frequency*> at the map node m.

C_{11} = {<cracker, 1>, <beer, 1>, <chicken, 1>, <pizza, 1>}
C_{21} = {<coke, 2>, <cracker, 2>, <beer, 1>}
C_{31} = {<beer, 1>, <chicken, 2>, <coke, 2>}

From all C_{m1}, reduce nodes collect and compute C_1 that is size 1 frequent item pairs and L_1 that is size 1 frequent item pairs that meet minimum support. Thus, *[pizza, 1]* of C_1 is eliminated in L_1.

C_1 = {[cracker, 3], [beer, 3], [chicken, 3], [pizza, 1], [coke, 4]};
L_1 = {[cracker, 3], [beer, 3], [chicken, 3], [coke, 4]}

(b) The second step
In the loop, L_1 is mapped to each map node m for L_{m1}.

L_{11} = {cracker, beer}
L_{21} = {chicken}
L_{31} = {coke}

Then, size 2 frequent item pair sets can be generated by joining and sorting L_1 to each item sets of the map node m as $C_{m2} = L_1$ *join_sort* L_{m1} where the duplicated item sets are eliminated:

C_{12} = {<beer, cracker>, <chicken, cracker>, <beer, chicken>, <coke, cracker>, <beer, coke>}
C_{22} = {<chicken, cracker>, <beer, chicken>, <chicken, coke>}
C_{32} = {<coke, cracker>, <beer, coke>, <chicken, coke>}

From all C_{m2}, *reduce* nodes collect C_2 that is size 2 frequent item pairs.

C_2 = {<beer, cracker>, <chicken, cracker >, <beer, chicken>, <coke, cracker>, <beer, coke>, <chicken, coke>};

C2 is mapped to each map node as C_{m2} as follows:

C_{12} = {<beer, cracker>, <chicken, cracker >}
C_{22} = {<beer, chicken>, <coke, cracker>}
C_{32} = {<beer, coke>, <chicken, coke>}

Now, we can generate size 2 frequent items with an item pair set [*item, frequency]* at the node *m* that contains all transaction data as presented in Figure 4.2.

C_{12} = {[<beer, cracker>, 2], [<chicken, cracker>, 0]}
C_{22} = {[<beer, chicken>, 1], [<coke, cracker >, 2]}
C_{32} = {[<beer, coke>, 2], [<chicken, coke>, 2]}

From all C_{m2}, the reduce nodes collect and compute C_2 that is size 2 frequent item pairs and L_2 that is size 2 frequent item pairs that meet minimum support. Thus, *[<chicken, cracker >, 0]* and *[<beer, chicken >, 1]* are eliminated from C_2 for L_2:

C_2 = {[<beer, cracker>, 2], [<chicken, cracker >, 0], [<beer, chicken >, 1], [<coke, cracker>, 2], [<beer, coke>, 2], [<chicken, coke>, 2]};
L_2 = {[<beer, cracker>, 2], [<coke, cracker>, 2], [<beer, coke>, 2], [<chicken, coke>, 2]};

(c) The third step
In the loop, L_2 is mapped to each map node *m*.

L_{12} = {<beer, cracker>, <coke, cracker>}
L_{22} = {<beer, coke>}
L_{32} = {<chicken, coke>}

Then, size 3 frequent item pair sets can be generated by joining and sorting L_2 to each item sets of the map node *m* as $C_{m3} = L_2$ *join_sort* L_{m2} where the duplicated item sets are eliminated:

C_{13} = {<beer, coke, cracker>, <beer, chicken, coke>, <chicken, coke, cracker>}
C_{23} = {<beer, coke, cracker>, <beer, chicken, coke>}
C_{33} = {<beer, chicken, coke>, <chicken, coke, cracker>}

As the item size *k* is or greater than *3*, *prune*(C_{mk}) deletes non frequent item sets that violates *apriori property* that all subsets of a frequent item set must also be frequent:

C_{13} = {<beer, coke, cracker>}
C_{23} = {<beer, coke, cracker>}
C_{33} = {}

C_{13}, C_{23}, C_{33} eliminate <*beer, chicken, coke*> and C_{13}, C_{33} removes <*chicken, coke, cracker*> as <*beer, chicken*> and <*chicken, cracker*> respectively are not a member of L_2, that is, non-frequent items.

From all C_{m3}, reduce nodes collect C_3 that is size 3 frequent item pairs and L_3 that is size 3 frequent item pairs that meet minimum support:

C_3 = {<beer, coke, cracker>}
L_3 = {}

Since L_3 does not have any element, the algorithm ends. Therefore, we have frequent item sets L_1 and L_2 with size 1 and 2 respectively:

L1 = {[cracker, 3], [beer, 3], [chicken, 3], [coke, 4]}
L2 = {[<beer, cracker>, 2], [<coke, cracker>, 2], [<beer, coke>, 2], [<chicken, coke>, 2]};

The item sets L_1 and L_2 can be used to produce association rule of the transaction.

5 Conclusion

The paper proposes *Apriori-Map/Reduce Algorithm* and illustrates its time complexity, which theoretically shows that the algorithm gains much higher performance than the sequential algorithm as the map and reduce nodes get added. The item sets produced by the algorithm can be adopted to compute and produce Association Rule for market analysis.

The future work is to build the code following the algorithm on Hadoop frame and generate experimental data by executing the code with the sample transaction data, which practically proves that the proposed algorithm works. Besides, the algorithm should be extended to produce association rule.

6 Reference

[1] "MapReduce: Simplified Data Processing on Large Clusters", Jeffrey Dean and Sanjay Ghemawa, Google Labs, pp. 137–150, OSDI 2004

[2] Apache Hadoop Project, http://hadoop.apache.org/,

[3] "Building a business on an open source distributed computing", Bradford Stephens , Oreilly Open Source Convention (OSCON) 2009, July 20-24, 2009, San Jose, CA

[4] "MapReduce Debates and Schema-Free", Woohyun Kim, Coord, March 3 2010

[5] "Data-Intensive Text Processing with MapReduce", Jimmy Lin and Chris Dyer, Tutorial at the 11th Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL HLT 2010), June 2010, Los Angeles, California

[6] "SQL MapReduce framework ", Aster Data, http://www.asterdata.com/product/advanced-analytics.php

[7] Apache HBase, "http://hbase.apache.org/"

[8] "Data-Intensive Text Processing with MapReduce", Jimmy Lin and Chris Dyer, Morgan & Claypool Publishers, 2010.

[9] GNU Coord, http://www.coordguru.com/

[10] "The Technical Demand of Cloud Computing", Jongwook Woo, Korean Technical Report of KISTI (Korea Institute of Science and Technical Information), Feb 2011

[11] "Market Basket Analysis Example in Hadoop, http://dal-cloudcomputing.blogspot.com/2011/03/market-basket-analysis-example-in.html", Jongwook Woo, March 2011

[12] Jongwook Woo, Siddharth Basopia, Yuhang Xu, Seon Ho Kim, "Market Basket Analysis Algorithm with NoSQL DB HBase and Hadoop", The Third International Conference on Emerging Databases (EDB 2011), Korea, Aug. 25-27, 2011

[13] Jongwook Woo and Yuhang Xu, "Market Basket Analysis Algorithm with Map/Reduce of Cloud Computing", The 2011 international Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2011), Las Vegas, July 18-21, 2011

Hybrid Algorithms for Matrix Multiplication on Multicore Clusters

Fabiana Leibovich, Marcelo Naiouf, Laura De Giusti, Fernando G. Tinetti[1] and Armando De Giusti

Instituto de Investigación en Informática LIDI (III-LIDI), Facultad de Informática, Universidad Nacional de La Plata, 50 y 120 2do piso, La Plata, Argentina.

Abstract - *Hybrid programming (through messages and shared memory) has gained importance since the appearance of multicore cluster architectures, fruit of the technological advance of processors and the physical limitations imposed by traditional architectures. This new programming paradigm allows exploiting the new memory hierarchy offered by the architecture.*
The purpose of this work is to carry out a comparative analysis between two hybrid algorithms that use two different parallel programming libraries for shared memory (PThreads and OpenMP). Both algorithms use MPI as message passing mechanism. For the experiments, the classic matrix multiplication problem is used, which is the basis for numerous applications.
The test architecture used for the experimental analysis is a multicore cluster. The performance obtained and programming tools are compared.

Keywords: parallel architectures, hybrid programming, cluster, multicore, message passing, shared memory.

1 Introduction

Parallel architectures have evolved to offer better response times for applications. As part of this evolution, clusters, then multi-cores, and currently multi-core cluster architectures, can be mentioned. The latter are basically a collection of multi-core processors interconnected through a network.

Multicore clusters allow combining the most distinctive features of clusters (use of message passing in shared memory) and multicores (use of shared memory). Also, they introduce modifications in memory hierarchy and further increase computer system capacity and power.

Taking into account the popularity of this architecture, it is important to study new parallel algorithms programming techniques that efficiently exploit its power, considering the hybrid systems in which shared memory and distributed memory are combined [1].

When it comes to implementing a parallel algorithm, it is very important to consider the memory hierarchy available, since this will directly affect algorithm performance. Figure 1 below shows the evolution.

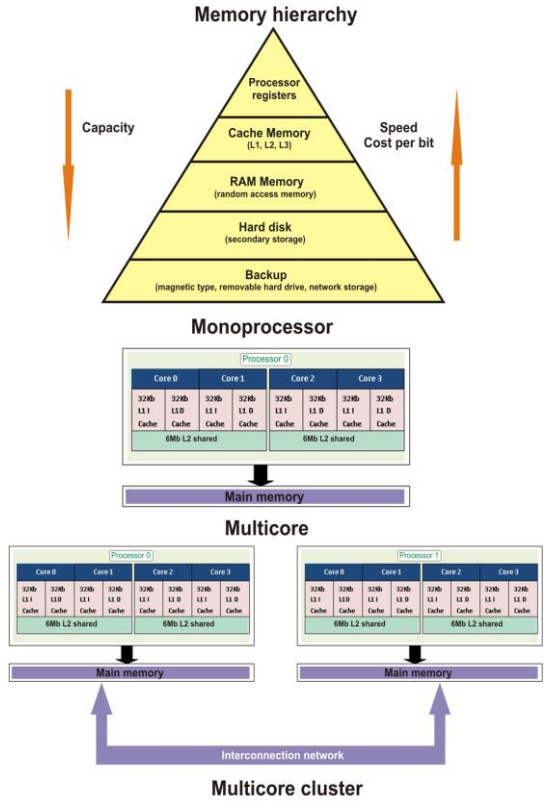


Figure (1) Memory hierarchy

Memory hierarchy performance is primarily determined by two hardware parameters: memory latency (time elapsed from the moment a piece of data is required and the moment it becomes available) and memory bandwidth (the speed with which data are sent from the memory to the processor). In the case of traditional clusters (both homogeneous and heterogeneous), there are memory levels in each processor (processor register and cache levels L1 and L2), but a new level is also included: network-distributed memory.

When considering a multi-core architecture, there are, in addition to register and L1 levels corresponding to each

[1]Investigador Comisión de Investigaciones Científicas Prov. de Bs. As.

core, two memory levels: cache memory shared by pairs of cores (L2) and memory shared among the cores of the multi-core processor [2]. Currently, the new architectures include one additional memory level, L3 cache.

In particular, multi-core clusters introduce one additional level to the traditional memory hierarchy. In addition to the cache memory shared between pairs of cores and the memory shared among all cores within the same physical processor, there is the distributed memory that is accessed through the network. This can also be seen in Figure 1.

This paper is organized as follows: Section 2 presents the objectives, then, Section 3 analyzes the parallel programming libraries used for the experiments. In Section 4, the existing mapping techniques used in this paper are discussed, while Section 5 presents the study case selected for testing and the solutions implemented. Finally, Section 6 discusses the results achieved, and Section 7 details the conclusions and future work.

2 Objective

The purpose of this paper is to carry out a comparative analysis of two hybrid solutions [3][4][5] that use the two shared memory libraries most commonly used nowadays in parallel computing: Pthreads and OpenMP.

Even though there is a large number of parallel applications in various areas, one of the most traditional and widely studied in parallel computation, and used in this paper, is matrix multiplication. It allows analyzing application scalability in two ways: by increasing the size of the problem and increasing the number of execution cores.

3 Parallel Programming Libraries

3.1. Pthreads

Over time, hardware manufacturers have implemented their own versions for thread management and administration. These versions are very different from each other, which makes it difficult for programmers to develop multithread applications that are portable. For this reason, in 1995 the IEEE established the POSIX (Portable Operating System Interface) standard. Its last version is IEEE Std 1003.1, 2004 Edition [6].

Pthreads is a library that implements the POSIX standard defined by IEEE, and is composed by a set of types and calls to procedures in programming language C that includes a header file and a thread library that is part, for example, of the libc library, among others. It is used for programming parallel applications that use shared memory.

The subroutines that form the API in Pthreads can be classified in four large groups: thread management, mutex (routines that handle synchronization and mutual exclusion), condition variables, and synchronization.

3.2. OpenMP

OpenMP is a programming interface (API) defined by a set of hardware and software manufacturer including the following: Sun Microsystems, IBM, Intel, AMD, among others [7].

It provides a portable and scalable interface for the developers of parallel applications that use shared memory.

This API supports C/C++ and Fortran in multiple architectures, including LINUX and Windows NT. It provides several builders and directives to specify parallel regions, shared work, synchronization, and environment variables.

It is formed by the following three components:

1. Compilation directives
2. Running time routine library
3. Environment variables

3.3. OpenMPI

MPI is a message passing interface that defines both the syntax and the semantics of the set of routines that can be used in the implementation of programs that use message passing. It was created to solve the issues that appeared after each hardware manufacturer defined its own communication interface, which in general were incompatible with all others. The purpose of MPI is to solve this problem by defining a standard [8].

MPI is a library that can be used to develop programs that use message passing (distributed memory) and uses the programming languages C or Fortran.

One of the implementations of this standard is OpenMPI, which is used in this paper, because it provides manual mapping mechanisms to assign processes to cores for their execution. This is discussed in Section 4.

3.4. Pthreads vs. OpenMP

It should be noted that *OpenMP* generates a pool of n threads (n is defined by the programmer) that is going to be re-used as needed in a very efficient manner. Thus, there is no need to create and remove threads when entering/leaving each parallel section. This is extensively exploited when the algorithm has several parallel sections.

In the case of Pthreads, threads would have to be created and destroyed for each of these sections, or a pool of threads would have to be created and kept from the algorithm. Also, if the algorithm required synchronization among threads, or more complex

parallelizations, such as shared variables reduction operations, combination of critical and non-critical sections, etc., OpenMP would in theory yield better results than Pthreads, since the library itself provides specialized and optimized mechanisms to do so, whereas Pthreads would require a manual implementation using semaphores or condition variables.

For this reason, one of the advantages of Pthreads over OpenMP is that the programmer has greater control over thread creation, destruction and behavior, since thread management is at a lower level than in OpenMP.

4 Thread and Process Manual Mapping

The traditional operating systems map threads and processes to processors and cores using various scheduling techniques.

However, there are mechanisms that allow application programmers to manually map these threads and processes. In the case of threads, the code of the application has to be modified, whereas in the case of process mapping, it can be done while it is running by using directly the binary.

Based on the architecture used in this paper, that is, multicore cluster, both mapping alternatives were used and combined to obtain a better global system performance. This can be done because the algorithms being studied use a hybrid model, combining shared memory and message passing.

4.1. Process Mapping

Open MPI is an open code project that implements the MPI standard and offers the added functionality of providing directives for explicitly mapping processes to processing cores. To do so, it requires two files called rankfile and hostfile.

The hostfile file defines the number of cores available in the system and the name of the machine in the network. The format of this file is the following:

hostNameX slots = number of cores
hostNameY slots = number of cores

The rankfile file defines an input for each process, as shown below:

rank N = hostNameX slot = number of CPU
rank M = hostNameY slot = number of CPU:number of core

Figure 2 shows a diagram explaining the parameters number of CPU and number of core.

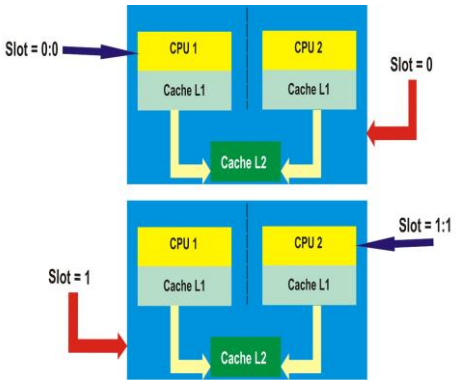


Figure (2) Manual mapping

These two files must be passed as parameter upon execution.

The advantage of this alternative is that the source code is untouched, there is no need to modify it to add this functionality, and as a consequence, the mapping process does not consume execution time. The disadvantage is that mapping is static and cannot be dynamically modified because it is passed as parameter upon execution.

4.2. Thread Mapping

In shared memory systems there is no specific function to assign a thread to a given processing core. Instead of this, there is a function that allows defining the affinity for each task.

Affinity allows specifying which, of all existing cores in a system, can be assigned a certain process or thread. To be able to assign these processes or threads to any given specific core, the core in question must be set as the only schedulable processor/core for such assignment.

Unix-based operating systems offer a system call that allows explicitly defining affinity. This should be used in shared memory environments, since it can manage the cores and processors in the machine on which the operating system is being run. The heading of this function is as follows:

sched_setaffinity(pid_t pid, unsigned long cpusetsize, cpu_set_t *mask)

Where:

- pid: is the process identifier for which affinity is defined. If its value is 0, it represents the process being run. It should be noted that the threads belonging to a same process share the same PID; however, the affinity belongs to each separate thread and is stored in the thread context.

- cpusetsize: this is the length (in bytes) of the data noted by the mask parameter.

- mask: this represents the affinity mask. It is a bit mask where each bit represents a (logical) processor in the system. Bits are sorted from the less significant one, corresponding to the first logical processor, to the most significant one, corresponding to the last logical processor in the system.

  If bit = 0, the processor is non-schedulable

  If bit = 1, the processor is schedulable

By default, when a thread/process is created, it is schedulable for all existing cores in the system. To map a process or thread to a specific processor, such processor must be set as the only schedulable processor.

One of the advantages of this scheduling technique is the possibility of dynamically modifying (during the execution) the schedule of any given thread/process. This is possible because the system call can be included in the application code.

5 Study Case

Given two matrixes A and B having dimensions m*p and p*n elements respectively, the multiplication of both matrixes consists in obtaining matrix C of dimension m*n elements (C = A*B), where each element is calculated with equation (1):

$$C_{i,j} = \sum_{k=1}^{p} A_{i,k} * B_{k,j} \qquad (1)$$

The implemented solutions can be classified in two classes: traditional matrix multiplication (I), and block-based matrix multiplication (II) (matrix C is calculated in blocks). In both cases, the implemented solutions are hybrid, combining shared memory with message passing, and use the master/worker interaction pattern. These were both developed using C language – in the case of message passing, the OpenMPI library [8] was used, while Pthreads [6] was used in one case of shared memory and OpenMP [7] was used in the other.

The testing architecture used should be noted, since it affects the implementation of the algorithms. The hardware used to carry out the tests was a Blade with 16 servers (blades). Each blade has 2 quad core Intel Xeon e5405 2.0 GHz processors; 2 Gb of RAM memory (shared between both processors); 2 X 6Mb L2 cache shared between each pair of cores by processor. The operating system used is Fedora 12, 64 bits [9][10].

5.1. Traditional Matrix Multiplication

There is one process per blade and each blade has 8 execution units – 7 threads are used for processing activities, added to the processing activities from the process itself that acts as a worker (one thread per core). A master/worker structure is used, with one of the processes acting as master, dividing the rows equally among all processes. Once this is done, it generates the corresponding processing threads (acting as worker). The other worker processes act in a similar way and send their results to the master process.

The algorithm can be summarized as follows:

Master process:

It divides the matrix into blocks of n rows/number of blades used for processing

It communicates the corresponding rows from matrix A and all of matrix B to worker processes.

It generates the threads and processes its own block

It receives results from worker processes.

Worker processes

They receive the corresponding rows from matrix A and all of matrix B.

They generate the threads to process the data.

They communicate the results to the master process.

Figure 3 shows an explanatory illustration. The numbers placed on matrix C indicate the number of the thread processed by each cell.

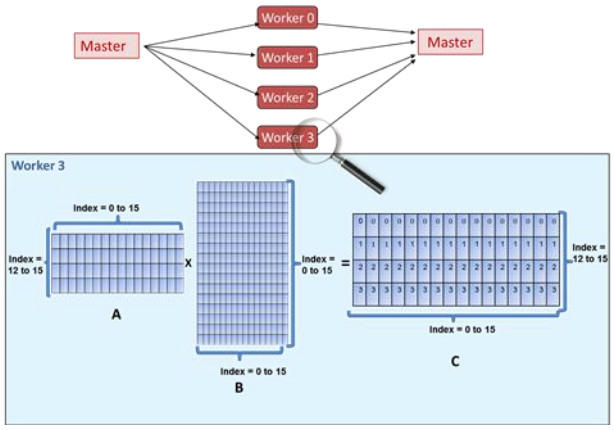


Figure (3) Classic matrix multiplication

5.2. Block Matrix Multiplication

Matrix C is calculated in blocks. To do so, each process receives the rows from matrix A and the columns from B required for calculating the block of matrix C that was assigned to it. The number of blocks into which matrix C is divided is divisible by the number of processes. The algorithm uses a master/worker-type interaction, where the master works both as coordinator and as worker. It divides matrix C into blocks to be processed and then generates processing phases. The same as in the traditional solution, each process is assigned to a blade and each thread to a core within the corresponding blade. Figure 4 shows an explanatory illustration. The numbers placed on matrix C indicate the number of the thread processed by each cell.

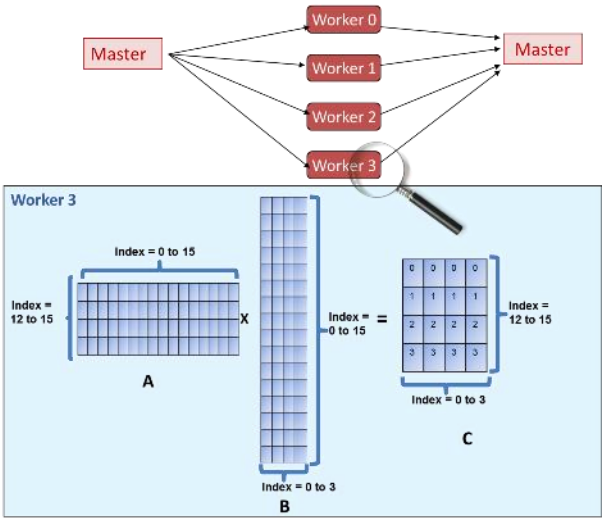


Figure (4) Block Matrix multiplication

Since all processors have the same computation power and all blocks to be processed are the same size, they will all be processed at (approximately) the same speed. Thus, for each processing phase, the master distributes the rows from matrix A and the columns from matrix B based on the block corresponding to each worker, including a block for itself. The master then processes its block and receives all the other results to move on to the next processing phase. The number of processing phases is calculated as follows: if b is the number of blocks that have to be processed and w is the number of workers (including the master, who also operates as a worker), the number of phases is b/w.

It should be noted that each process must store the rows from matrix A to be processed, the columns from matrix B, and the block from matrix C that it generates as a result.

In each phase, once the master distributes the blocks to the workers, it generates the corresponding threads to process its own block, dividing it into rows for each thread to process a subset of rows. The other worker processes act in a similar way, receiving data and sending their results to the master process.

The algorithm can be summarized as follows:

Master process:
It divides matrix C into blocks.
For each phase:
It communicates the corresponding rows from matrix A and the corresponding columns from matrix B to worker processes, based on the block assigned to each of them.
It generates the threads and processes its own block.
It receives results from worker processes.
Worker processes
For each phase:
They receive the data to be processed
They generate the threads and process the data.
They communicate the results to the master process.

6 Results obtained

The following tables show the execution times (expressed in seconds), the difference in time between both implementations, and the percent difference, (based on *Pthreads*) obtained for solution (I) explained in the previous section for 16 (Table 1) and 32 (Table 2) cores, and the results obtained by solution (II) for 16 (Table 3) and 32 (Table 4) cores.

Also, two charts are shown comparing solution execution times for traditional multiplication (Figure 5) and block multiplication (Figure 6).

Size	Pthreads	OpenMP	Difference	Percentage
1024	0.32	0.32	0.0073	2.29
2048	2.28	2.34	0.0527	2.30
4096	17.19	17.51	0.3115	1.81
8192	133.18	139.59	6.41	4.81
9600	216.73	213.11	-3.61	-1.66

Table (1)

Size	Pthreads	OpenMP	Difference	Percentage
1024	0.22	0.23	0.015	7.14
2048	1.40	1.42	0.014	1.06
4096	9.56	9.76	0.198	2.07
8192	69.91	74.15	4.23	6.06
9600	113.48	113.51	0.027	0.02

Table (2)

Size	Pthreads	OpenMP	Difference	Percentage
1024	0.16	0.17	0.0072	4.39
2048	1.59	1.60	0.0047	0.29
4096	11.97	11.87	-0.0981	-0.81
8192	101.41	104.37	2.9637	2.92

Table (3)

Size	Pthreads	OpenMP	Difference	Percentage
1024	0.17	0.17	0.0037	2.16
2048	1.15	1.12	-0.0286	-2.47
4096	7.37	7.37	0.0069	0.093
8192	55.47	58.13	2.6632	4.800

Table (4)

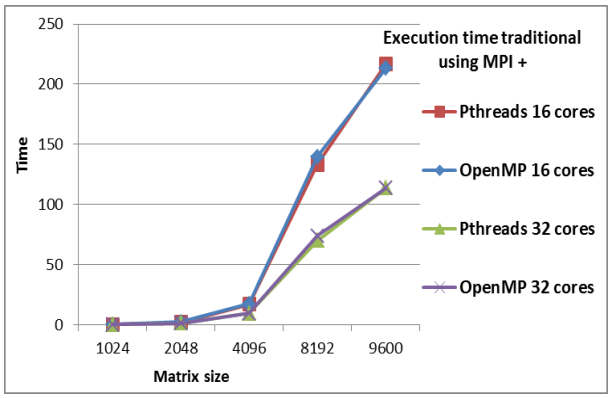


Figure (5)

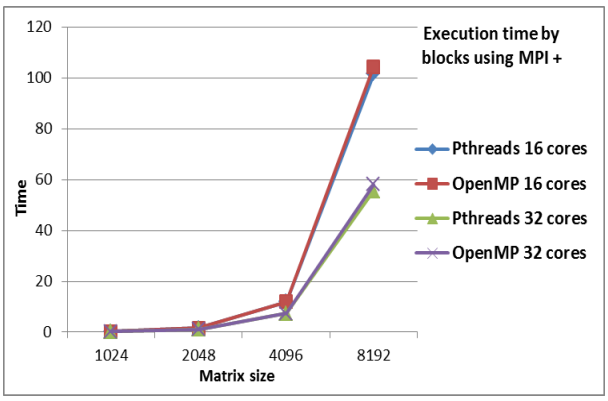


Figure (6)

7 Conclusions and future work

As it can be observed, the difference in the times obtained by both implementations is not significant – it is never greater than 10%, – but in most of the cases for this problem, *Pthreads* yields slightly better results than *OpenMP*.

According to a report published by Intel, *OpenMP* has an initial *overhead* in its primitives as in the case of *#pragma omp parallel for*. This *overhead* will be significant or not depending on how much it represents, percentage-wise, in relation to the total execution time of the algorithm. It should be noted, however, that this *overhead* is initial and is not related to the operations executed later on [11].

In algorithms that efficiently exploit optimized operations of *OpenMP*, the difference between running the same algorithm with *Pthreads* or with *OpenMP* will in principle be more significant in favor of the latter [12].

However, from a programming effort standpoint, OpenMP markedly simplifies algorithm implementation thanks to the directives it provides. It allows parallelizing a sequential solution in only a few steps, given its high abstraction level. In fact, a solution that has been parallelized with OpenMP allows running the algorithm as if it were the original sequential solution just by disabling the library. The abstraction level provided by OpenMP greatly facilitates the learning and use of the library. Also, the programming directives themselves document the parallelization of the application.

As future line of work, we will work in adapting the algorithms to be run on a heterogeneous architecture, both from the point of view of processor speed and memory hierarchy. Thus, the algorithms will have to respond to new challenges such as load balancing and dynamic work distribution, among others.

8 References

[1] Dongarra J. , Foster I., Fox G., Gropp W., Kennedy K., Torzcon L., White A. "Sourcebook of Parallel computing". Morgan Kaufmann Publishers 2002. ISBN 1558608710 (Chapter 3).

[2] Burger T. "Intel Multi-Core Processors: Quick Reference Guide http://cachewww.intel.com/cd/00/00/23/19/231912_231912.pdf. (2010).

[3] Kumar V., Gupta A., "Analyzing Scalability of Parallel Algorithms and Architectures". Journal of Parallel and Distributed Computing. Vol 22, No.1.pp 60-79. 1994.

[4] Leopold C., "Parallel and Distributed Computing. A Survey of Models, Paradigms and Approaches". Wiley, 2001. ISBN: 0471358312 (Chapters 1, 2 and 3).

[5] Chapman B., "The Multicore Programming Challenge, Advanced Parallel Processing Technologies"; 7th International Symposium, (7th APPT'07), Lecture Notes in Computer Science (LNCS), Vol. 4847, p. 3, Springer-Verlag (New York), November 2007.

[6] https://computing.llnl.gov/tutorials/pthreads

[7] https://computing.llnl.gov/tutorials/openMP

[8] http://www.open-mpi.org

[9] HP, "HP BladeSystem". http://h18004.www1.hp.com/products/blades/components/c-class.html. (2011).

[10] HP, "HP BladeSystem c-Class architecture". http://h20000.www2.hp.com/bc/docs/support/SupportManual/c00810839/c00810839.pdf. (2011).

[11] Lindberg, P., "Basic OpenMP Threading Overhead". Intel.

[12] Furlinger, K., Gerndt, M. Analyzing Overheads and Scalability Characteristics of OpenMP Applications.

Modeling of Hierarchical Multiprocessor Database Systems*

Pavel S. Kostenetskiy, Leonid B. Sokolinsky
South Ural State University, Chelyabinsk, Russia

Abstract - *The paper is devoted to the issues concerning with modeling and simulating of hierarchical database multiprocessor systems. The requirements for a model of parallel database systems are defined. A new computational model of the hierarchical database multiprocessor architecture is described. This model is called DMM (Database Multiprocessor Model) and designed for OLTP workload. It allows us to simulate and analyze an arbitrary multiprocessors configuration for database application. The practical experience of exploiting the DMM model is discussed.*

Keywords: parallel query processing, database multiprocessor model.

1 Introduction

Modern multiprocessor systems mostly have a hierarchical topology. So, the modern computing clusters have three-level multiprocessor architecture. The multicore processor forms the first level of hierarchy. Each computing node of the cluster is a multiprocessor with shared memory. It is the second level of hierarchy. The third level of hierarchy is presented by computing cluster consisting of homogeneous computing nodes interconnected by high-speed network. Another source of multiprocessor hierarchies is presented by Grid-technologies. Grid-system can consist of several clusters interconnected by LAN. Grid-systems can be divided into three classes in accordance with communication network size: intragrid, extragrid and intergrid. Intragrid system incorporates the set of clusters interconnected by LAN inside one organization. Extragrid system is built on interconnection of several intragrid systems. Intergrid system can integrates dozens of extragrid systems in united computational meta-system by the Internet. The general structure of the multiprocessor hierarchy is shown in Fig. 1.

In the nearest future, big organizations will exploit databases which contain many petabytes of information [2]. The hierarchical multiprocessor multicore systems with hundred thousand processors will be needed to manage such huge volumes of data. Therefore the issues of mathematical modeling, simulating and analyzing of the new hierarchical database multiprocessors architectures are important. The first papers devoted to parallel database modeling are the following [1, 3, 4, 5, 6, 7, 8]. All the models proposed in these papers took into account the peculiarity of database

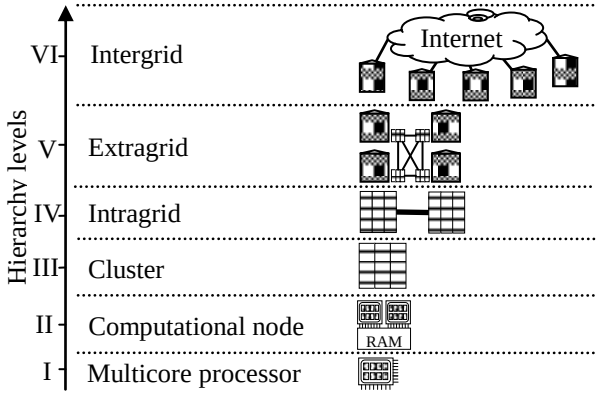


Fig. 1. Structure of multiprocessor hierarchy

applications. However, all these models ignored communication overhead, which is very important for parallel database systems. The first work taking into account the communication overhead was the paper [10]. In this paper, the model of Gamma database machine [9] for multiprocessor system InteliPSC/2 Hipercube consisting of 32 processor nodes was described. The modeling language DeNet [12] was used for description of the model. The model had a network structure. The nodes were presented by abstract modules generating discrete events. The arcs represented the data flows. The certain message type could be associated with each arc.

The further development in modeling of parallel database systems is related with appearance of the hierarchical systems. In paper [14], the model of two-level architecture was described. This model included the transaction, system and queue submodels. In accordance with classification described in work [13], this architecture can be classified as CE (*Clustered Everything*). In such a case, the system consists of *N shared everything* clusters interconnected by the global communication network in *shared nothing* manner.

The database is simulated as a collection of relations, each of which is uniformly partitioned between disks. For every attribute, the set of attribute values is simulated by means of specifying the probability distribution function and the number of unique attribute values. It is supposed that distribution function can be uniform or normal. In addition, it is expected that the attribute distribution function is the same for all the disks and every unique value is presented on every disk. There are two types of transactions in the transaction model: 1) update transactions, 2) query transactions.

Update transactions are simulated as a sequence of the read/write page operations. Query transactions correspond to

* Supported by the Russian Foundation for Basic Research under Grant 12-07-00443-a and by the Federal Program 07.514.11.4036.

relational operators. They are simulated as a sequence of read operations, because the intermediate result is saved in the main memory or in the temporary file and doesn't change the database. Each transaction is represented by the set of processes that run on different nodes and use data interchange. The proposed model doesn't support the data replication. It doesn't also support Zipf's distribution and 80/20 rule. Moreover, this model does not admit a generalization on the arbitrary number of hierarchy levels.

The remainder of the paper has the following outline. In Section 2, the new computational model for hierarchical database multiprocessor architecture is described. This model is called *DMM* (Database Multiprocessor Model). In Section 3, the results of simulating database applications in various multiprocessor configurations are proposed. The last section gives summary of the basic results obtained and conclusions, as well as discusses directions of future research.

2 Database Multiprocessor modules

The *DMM* model includes the following sub-models: *hardware model*, *software model*, *cost model* and *transaction model*.

2.1 Hardware model

The set of multiprocessor system modules is divided into three non-overlapping subsets:

$$\mathfrak{M} = \mathfrak{P} \cup \mathfrak{D} \cup \mathfrak{N},\ \mathfrak{P} \cap \mathfrak{D} = \varnothing,\ \mathfrak{P} \cap \mathfrak{N} = \varnothing,\ \mathfrak{D} \cap \mathfrak{N} = \varnothing.$$

There, $\mathfrak{P}$ is the set of *processor modules*, $\mathfrak{D}$ is the set of *disk modules* and $\mathfrak{N}$ is the set of *hub modules*. The *DMM* model does not provide a representation of memory modules, since in *OLTP* workload, the disk access time is much more then memory access time.

The *DM-graph* is connected acyclic graph $\mathfrak{W}(\mathfrak{M}, \mathfrak{E})$, where the set of edges $\mathfrak{E}$ meets the following restrictions:

$$\mathfrak{P} \cup \mathfrak{D} \neq \varnothing, \mathfrak{N} \neq \varnothing; \quad (1)$$

$$\forall P \in \mathfrak{P}\left(\forall M \in \mathfrak{M}\left(\left(\exists E(A,A') \in \mathfrak{E}\begin{pmatrix}(\text{init}(A) = P \wedge \text{fin}(A) = M)\\ \vee(\text{init}(A') = P \wedge \text{fin}(A') = M)\end{pmatrix}\right) \Rightarrow M \in \mathfrak{N}\right)\right); \quad (2)$$

$$\forall D \in \mathfrak{D}\left(\forall M \in \mathfrak{M}\left(\left(\exists E(A,A') \in \mathfrak{E}\begin{pmatrix}(\text{init}(A) = D \wedge \text{fin}(A) = M)\\ \vee(\text{init}(A') = D \wedge \text{fin}(A') = M)\end{pmatrix}\right) \Rightarrow M \in \mathfrak{N}\right)\right); \quad (3)$$

$$\forall M \in \mathfrak{M}\left(\left(\forall E(A,A'), \tilde{E}(\tilde{A},\tilde{A}')\left(\left(\begin{pmatrix}(\text{init}(A) = M \vee \text{init}(A') = M)\\ \wedge(\text{init}(\tilde{A}) = M \vee \text{init}(\tilde{A}') = M)\end{pmatrix} \Rightarrow (E = \tilde{E})\right)\right) \Rightarrow (M \in \mathfrak{D} \cup \mathfrak{P})\right). \quad (4)$$

Condition (1) means that simulating hardware model must include at least one hub and at least one processor or disk. Conditions (2) and (3) mean that processor and disk modules are able to connect only with hub modules. Condition (4) means that disk and processor modules may not have any descendants, and thus are always the leaf nodes of *DM*-tree.

DM-tree is the *DM*-graph with dedicated node $\bar{N} \in \mathfrak{N}$ called root hub.

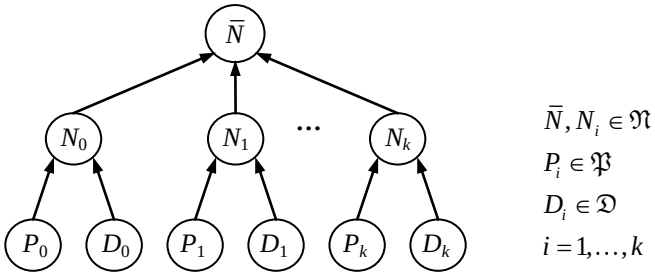


Fig. 2. *DM*-tree example

2.2 Software model

The smallest unit of measuring data in the *DMM* is defined as a "*packet*". We will assume that all packets have the same size. The header of packet includes sender address, the recipient address and some other information. Transfer of a packet corresponds to transfer of one or several tuples in the real databases system.

The *DMM* model does not provide a representation of memory modules, since in *OLTP* workload, the disk access time is much more then memory access time. Any processor module can exchange data with any disk module. Disk modules and hub modules have queue buffers, which are used for packet transfer. The *DMM* model assumes the asynchronous transfer mode. It means that the processor module may start a new exchange without waiting the completion of previous one. However, we suppose that processor module may have not more than s_r uncompleted read operations and not more than s_w uncompleted write operations.

In the *DMM* model, data processing is divided into discrete time intervals called *ticks*. The tick can be defined as a predetermined sequence of steps, which will be described below.

Let $M \in \mathfrak{M}$. We will use the following denotations: $F(M)$ – parent module of node M, $T(M)$ – subtree, which has M as a root.

Processor module $P \in \mathfrak{P}$ may activate read/write operations. Let's define their semantic as follows.

Read Operation. Let processor module P has to read packet E from disk module $D \in \mathfrak{D}$. If P has already activated s_r uncompleted read operations then P has to be suspended. Otherwise, the packet E has to be put in the queue buffer of disk D. At that case, packet E has $\alpha(E)=P$ as a recipient address and $\beta(E)=D$ as a sender address. This algorithm is shown in Fig. 3.

```
if r(P) < s_r then
  Put packet E with address P in the
    queue buffer D
  r(P)++;
else
  wait;
end if
```

Fig. 3. Processor module read algorithm

* Supported by the Russian Foundation for Basic Research under Grant 12-07-00443-a and by the Federal Program 07.514.11.4036.

```
if w(P) < s_w then
  Put packet E in the queue buffer of
    parent hub module;
  w(P)++;
else
  wait;
end if
```

Fig. 4. Processor module write algorithm

There, $r(P)$ is the number of uncompleted read operations for processor module P, s_r is the maximum number of uncompleted read operations.

Write operation. Let processor module P has to write packet E to the disk module $D \in \mathfrak{D}$. The algorithm is shown in Fig. 4.

There, $w(P)$ is the number of uncompleted write operations for processor module P, s_w is the maximum number of uncompleted write operations.

Hub module $N \in \mathfrak{N}$ permanently transfers packets through interconnect. It performs an algorithm, which is shown in Fig. 5.

```
Remove packet E from queue of N;
if α(E) ∉ T(N) then
  Put E in queue F(N);
else
  Find maximum subtree U of
    T(N): α(E)∈U;
  if T(α(E))==U then
    if α(E)∈𝔓 then
      r(α(E))-- ;
    else
      Put E in queue α(E);
    end if
  else
    Put E in queue R(U);
  end if
end if
```

Fig. 5. Hub module algorithm

```
Remove packet E from queue of module D;
if α(E)∈𝔇 then
        w(β(E))-- ;
else
        Put E in the parent queue;
end if
```

Fig. 6. Disk module algorithm

There, E is a packet, $\alpha(E)$ is the recipient address of packet E, $T(N)$ is a subtree, which has N as a root, $F(N)$ is the parent module for N, $\mathfrak{P}$ is a set of all processor modules, $r(P)$ is the number of uncompleted read operations for processor module P, $R(U)$ is the root of a subtree U.

Disk module $D \in \mathfrak{D}$ permanently executes read/write operations. It performs an algorithm, which is shown in Fig. 6. There, $\beta(E)$ is a sender address, $w(\beta(E))$ is the number of uncompleted write operations of sender.

In the *DMM* model, data processing is divided into discrete time intervals called *ticks*. The tick can be defined as a predetermined sequence of the following steps:

1) each hub module handles all packets, which are waiting for transfer;
2) each active processor module performs one read or write operation;
3) each disk module handles one packet from its queue.

It is obvious that in this case, the queue of any hub module may contain not more then $|\mathfrak{P}|+|\mathfrak{D}|$ packets, and the queue of any disk module may contain not more then $s_r s_w |\mathfrak{P}|$ packets.

2.3 Cost model

Each module $M \in \mathfrak{M}$ has a *cost coefficient* $h_M \in \mathbb{R}$, $1 \le h_M < +\infty$. The time needed for a processor to process one packet for *OLTP* applications is roughly 10^5-10^6 times smaller, than the time it takes to transfer data to or from a hard disk, or from the network. Therefore, for all processor modules: $h_P = 1, \quad \forall P \in \mathfrak{P}$.

Hub module N can send more than one packet in each tick. Therefore the following interference function is associated with each hub module $N \in \mathfrak{N}$: $f_N(m_i^N) = \left\lceil \frac{m_i^N}{\tau_N} \right\rceil \delta_N$.

There, m_i^N – is the number of packets transferred by N in the i-th tick; $0 \le \delta_N \le 1$ – scale coefficient, $\tau_N > 1$ is threshold coefficient (the maximum number of the packets sending at the same time which doesn't slow hub module's work). Therefore, the time it takes for hub module N to process i-th tick can be calculated by the formula:

$$t_i^N = h_N f_N(m_i^N), \qquad \forall N \in \mathfrak{N}.$$

The total time spent by the system to process a mix of transactions over k ticks can be calculated by the formula: $t = \sum_{i=1}^{k} \left(\max_{N \in \mathfrak{N}}(t_i^N) + \max_{N \in \mathfrak{D}}(h_D) \right)$.

2.4 Transaction Model

Serial transaction is the transaction which runs on one processor module. Serial transaction Z is simulated by setting of two process groups ρ and ω: $Z = \{\rho, \omega\}$, $\rho \cup \omega \neq \varnothing$. Group ρ includes *read* processes. Group ω includes *write* processes. These processes are an abstract representation of read/write operations performed during transaction processing. Each read and write process has to make the certain *number of disk accesses*. After making all the accesses the process is removed from ρ or ω respectively. Transaction $Z = \{\rho, \omega\}$ is considered as finished when $\rho \cup \omega = \varnothing$.

In the *DMM* model each transaction $Z = \{\rho, \omega\}$ is divided into the finite sequence of steps: $Z_1, \ldots, Z_s$. There, s is the number of transaction steps. In accordance with this each process $x \in \rho \cup \omega$ of transaction Z is divided into s steps: $x_1, \ldots, x_s$. Each step x_i $(i = 1, \ldots, s)$ is described by the set of three numbers (n_i, p_i, d_i). There, d_i is the disk number the process x exchanges data with, n_i is number of disk accesses, p_i is probability of access by process x to disk D_{d_i} on each tick of the simulator running during i-th step.

* Supported by the Russian Foundation for Basic Research under Grant 12-07-00443-a and by the Federal Program 07.514.11.4036.

Consider the following example. Let transaction $Z=\{\rho,\omega\}$ represent MHJ JOIN of two relations. Let's assume that build relation *R* occupies 10 blocks and might be fully placed in main memory, probe relation *S* occupies 10 000 blocks. Let the result be 1000 blocks. Assume that input relations are placed on disk D_5; result is also being saved on D_5. In this case transaction Z is divided into two steps: on the first step the hash table for build relation is being constructed (read process a); on the second step the scanning of probe relation (read process b) and saving the output relation to the disk (write process c) are occurring. Thus, the set of read processes is $\rho=\{a,b\}$, the set of write processes is $\omega=\{c\}$. Each process is divided into two serial steps.

For read process *a* the decomposition is $a=\{a_1,a_2\}$, where $a_1=(10,1,5)$, $a_2=(0,1,5)$. The process *a* makes ten accesses to disk D_5 with probability of 1 on the first step. This step simulates the constructing of hash table in main memory. On the second step process *a* doesn't make any accesses to the disk.

For read process *b* the decomposition is $b=\{b_1,b_2\}$, where $b_1=(0,1,5)$, $b_2=(10000,1,5)$. On the first step process *b* doesn't make any accesses to the disk. On the second step process *b* makes 10 000 accesses to disk D_5 with probability of 1. This step simulates the scanning of probe relation.

For write process *c* the decomposition is $c=\{c_1,c_2\}$, where $c_1=(0,1,5)$, $c_2=(1000,0.1,5)$. On the first step process *c* doesn't make any accesses to the disk. On the second step process *c* makes 1000 accesses to disk D_5. Herewith on each tick of simulator the probability of disk access is 0.1. The last number is obtained as result of dividing the number of blocks of build relation on the number of blocks of probe relation. We suppose here that values of join attribute are uniformly distributed. For simulation of nonuniform distribution (e.g. the 80/20 rule) it is necessary to use more steps.

Parallel transaction is the transaction that requires two or more processor modules for processing. Parallel transaction *Z* that is being processed on *l* processor modules is simulated as *l* serial transactions $Z^1,\ldots,Z^l$, each of which is processed on the separate processor module. So we can consider that only serial transactions are being processed in the system.

The algorithm that simulates the processing of one transaction on the processor module is described below. Let transaction $Z=\{\rho,\omega\}$ consist of *s* steps $Z_1,\ldots,Z_s$. Let *X* be the set of all read and write processes that compose transaction *Z*: $X=\rho\cup\omega$. Let $X=\{x^1,\ldots,x^r\}$. Each process x^j ($j=1,\ldots,r$) is divided into *s* steps: $x_1^j,\ldots,x_s^j$. Suppose that the *i*-th step of transaction Z is started. Assume that the process x^j obtained the control. Let *i*-th step of process x^j be $x_i^j=(n_i^j,p_i^j,d_i^j)$. It's supposed that process x^j is able to obtain the control in the *i*-th step only if $n_i^j>0$. Then the following sequence of actions runs on the current tick.

1) The value of n_i^j decrements by one.
2) Process x^j initiates exchange operation with disk d_i^j.
3) If $\sum_{j=1}^{r} n_i^j=0$ and $i<s$, then increment i (current transaction step number) by one.

State $n_i^j=0$ corresponds to the ending of *i*-th step of process x^j. State $\sum_{j=1}^{r} n_i^j=0$ corresponds to the ending of *i*-th step of transaction *Z*. Thus, only when all the processes of transaction *Z* finished the processing of *i*-th step, the transaction transfers to step *i*+1.

DMM model allows the processing of *serial transaction mix* $\{Z^i\,|\,i=1,\ldots,k\}$ on one processor in the time-sharing mode. In this case each transaction Z^i $(i=1,\ldots,k)$ appears as its own pair of read and write processes group: $Z^i=\{\rho^i,\omega^i\}$. All the quantity of processes that simulate transaction mix processing on the processor $P\in\mathfrak{P}$ is defined by the following formula: $\Phi_P=\bigcup_{i=1}^{k}(\rho^i\cup\omega^i)$.

The read and write processes that describe the transaction are dynamically appended to the set Φ_P when the new transaction runs on the processor *P*. If some process is finished, it is being deleted from the set Φ_P.

Let's consider the processing of transaction mix Φ_P on processor *P* in the *DMM* model. Processor *P* must initialize one disk access operation on each tick. In accordance with this processor must choose the process $x\in\Phi$ and perform the read or write operation with disk that associated with *x* during the current step. We will call such process *active*. All the processes in the set Φ are organized in the list. Let the pointer to the current list element be (its initial position might be random). In the Fig. 7 the algorithm for defining the active process is shown.

```
//Loop on processors:
for (P = P.begin(); P != P.end(); P++){
   prob = 0; // Summary probability
   //Loop on processes:
   for(x= P.Φ.begin(); x!= P.Φ.end(); x++){
      if (n(x,i(x)) == 0) continue;
      prob += p(x,i(x));
      if (g(prob)) return x;
   };
};
```

Fig. 7. Active process *x* choosing procedure

There, $\mathfrak{P}$ is the set of all processor modules of *DM*-tree, *s(x)* is the number of process *x* steps, *n(x,i)* is the number of disk accesses that process *x* still has to make during *i*-th step, *p(x,i)* is the probability of process's *x* disk access during *i*-th step, *g* is the actuation function and defined by the following description. For each step *i* of process $x\in\Phi$ the probability of access by process *x* to disk associated with this process on *i*-th step is known. The actuation function $g(p_i^x)=G$ is

* Supported by the Russian Foundation for Basic Research under Grant 12-07-00443-a and by the Federal Program 07.514.11.4036.

described by the function of discrete random variable G, which defined by the following distribution law:

G	1	0
P	p_i^x	$1-p_i^x$

Let's review the *transactions creation* in *DMM* model using *MHJ* (*Main memory Hash Join*) algorithm as the example. This algorithm is intensively used in modern DBMS for query processing in cases when one of the relations is completely placed in main memory. MHJ algorithm is divided into "*build*" and "*probe*" *phases*. The hash table for the relation *R* is being constructed on the "*build*" *phase*. Herewith each processor node makes the following operations:

1. Sequential scanning of tuples of the own fragment of relation *R*. For each input tuple the distribution function ψ is calculated. This function calculates the number of the cluster node where this tuple has to be processed. The tuple is transferred to this node.
2. The hash table is built in the main memory of the node using hash function *h*(*t*) for tuples that received during step 1.

The processing of the relation *S* and joining with the tuples from relation *R* are made on the "*probe*" *phase*. Herewith each processor node makes the following operations:

1. Sequential scanning of tuples of the own fragment of relation *S*. For each input tuple the distribution function ψ is calculated. The tuple is transferred to the cluster node where this tuple has to be processed.
2. The tuple from step 1 is received and joined with tuples from step 2 of build phase. The output tuple is created.

Exchange operator redistributes the tuples among the processor nodes on each algorithm phase. The parallel query plan for MHJ operation is shown in Fig. 8. *Scan* operator scans the relation. *Exchange* operator is inserted as the left son and right son of MHJ operator and redistributes the tuples from *scan* operator among the processor nodes during the query processing.

Suppose that we need to create the model of the parallel transaction which is the natural join of *R* and *S* relations on their common attribute *A*. We suppose that the values of attribute *A* are uniformly distributed among fragments of relations *R* and *S*. For both relations we used the skew coefficient μ. According to this coefficient the tuples are divided into two classes: "own" and "alien". μ coefficient determines the percentage of "own" tuples in the fragment. The "own" tuples should be processed on the same cluster node where they are being stored. The "alien" tuples should be transferred to another cluster node on the basis of uniform distribution. For example, if $\mu = 0.5$ the relation is formed in such a way that each parallel agent passes to the other agents about 50% of the tuples of its fragment during the join processing. Herewith all the other agents receive about the same number of tuples.

We suppose that relation *R* is divided into equal-sized fragments $R_i\,(i = 0,\ldots, N-1)$. Each of these fragments is placed on the node which number coincides with the fragment number. Herewith any fragment R_k is able to be placed in main memory of the node. Similarly *S* is divided into equal-sized fragments $S_i\,(i = 0,\ldots, N-1)$. Each of these fragments is placed on the node which number coincides with the fragment number.

Parallel transaction is represented as the set of one-type serial transactions. Each of these transactions is processed on the distinct cluster nodes. Let the cluster node #0 process transaction $Z = \{p\}$. Transaction *Z* and all the processes occurring in it are divided into two steps. The first step simulates the build phase, the second step simulates the probe phase. The set of read processes of transaction *Z* is $\rho = \{a^0,\ldots,a^{N-1}, b^0,\ldots,b^{N-1}\}$.

Processes $a^0,\ldots,a^{N-1}$ conform to build phase. Process a^0 simulates the reading of "own" tuples of relation *R* from own disk D_0. It consists of two steps: $a^0 = \{a_1^0, a_2^0\}$, where $a_1^0 = (\frac{\mu |R|}{N}, \mu, 0)$, $a_2^0 = (0,0,0)$. Processes $a^j\,(j = 1,\ldots, N-1)$ simulate the reading of "own" tuples of relation *R* from alien disks. Each such process consists of two steps: $a^j = \{a_1^j, a_2^j\}$, where $a_1^j = (\frac{(1-\mu)|R|}{N^2 - N}, \frac{(1-\mu)}{(N-1)}, j)$, $a_2^j = (0,0,j)$.

Processes $b^0,\ldots,b^{N-1}$ conform to probe phase. Process b^0 simulates the reading of "own" tuples of relation *S* from own disk D_0. It consists of two steps: $b^0 = \{b_1^0, b_2^0\}$, where $b_1^0 = (0,0,0)$, $b_2^0 = (\frac{\mu |S|}{N}, \mu, 0)$. Processes $b^j\,(j = 1,\ldots, N-1)$ simulate the reading of "own" tuples of relation *S* from alien disks. Each such process consists of two steps: $b^j = \{b_1^j, b_2^j\}$, where $b_1^j = \{0,0,0\}$, $b_2^j = (\frac{(1-\mu)|S|}{N^2 - N}, \frac{(1-\mu)}{(N-1)}, j)$.

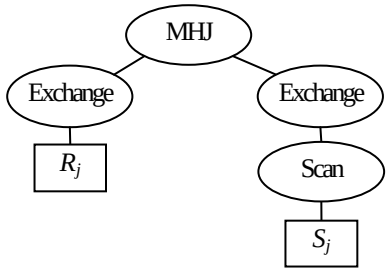


Fig. 8. MHJ query parallel plan

The transactions that are being processed on other cluster nodes are formed analogically.

3 DMM model usage

The *Database Multiprocessor Simulator* is developed on the basis of *DMM*. It allows to simulate and investigate various multiprocessor configuration for *OLTP* workload. The source

* Supported by the Russian Foundation for Basic Research under Grant 12-07-00443-a and by the Federal Program 07.514.11.4036.

code of the simulator is available on the Internet at http://kps.susu.ru/science/dms/sources/DMS-sources.zip. The simulation experiments for transaction processing on existing cluster configurations were performed with the simulator. Then these transactions were performed on the real cluster systems. They demonstrate that the simulator demonstrates the results that are identical to the result of real parallel DBMS processing on the cluster with the simulated architecture. It confirms the adequacy of *DMM*. In this section we are exemplifying the usage of simulator to evaluate the effectiveness of purchasing and upgrading decision of database oriented multiprocessor systems.

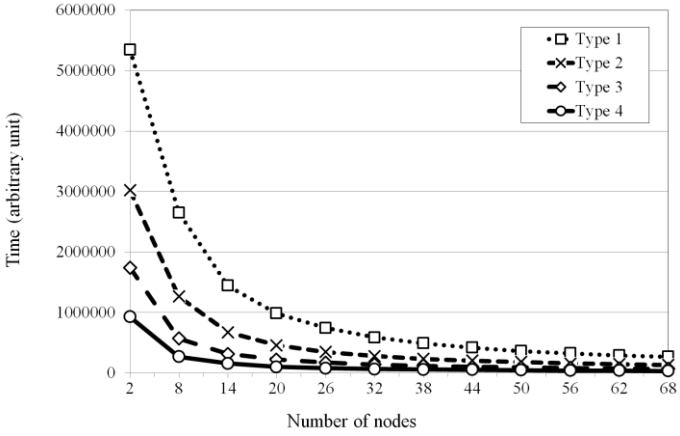


Fig. 9. The dependence of the query processing time from the number and type of nodes

In our experiments we processed natural join for relations *R* and *S* by the *MHJ*. The number of tuples of R and S, respectively, were 6 000 000 and 600 000 000 tuples.

The diagrams of dependence of query processing time from the number and type of nodes were obtained with emulator. They are shown in Fig. 9. The four types of nodes with characteristics and price given in Table 1 were simulated. The price consists of the price of infrastructure (UPS, cooling system, InfiniBand QDR communication network, cases, power supply units, racks, etc.).

Table 1. Prices of computing nodes

Node type	Main characteristics	Price (€)		
		year 2009	year 2011	year 2013 (forecast)
1	One single core CPU and one SAS disk	7,100	6,650	–
2	One double core CPU and two SAS disks	7,775	7,450	7,375
3	Two double core CPU and four SAS disks	8,725	8,125	7,750
4	Two quad-core CPU and eight SAS disks	10,425	9,875	8,650

Table 2. Performance of computing clusters

Number	Configuration	Transaction processing time (in arbitrary units)
1	37 type 1 nodes	496032
2	33 type 2 nodes	265325
3	30 type 3 nodes	146713
4	25 type 4 nodes	78520

Let's exemplify the usage of *DMS* simulator for solving the problems of long-term planning of development of computational capacities in data processing centers.

Example 1. Suppose that the organization has €250,000 for purchasing a computer cluster for database applications. We have to select the most productive hardware architecture, varying the configuration and number of nodes in the system. The cost of this architecture should not exceed that sum of money. Based on dependencies that shown in Fig. 9 we constructed Table 2. It contains different variants of cluster configurations, which cost about €250,000. The transaction processing time in arbitrary units is shown for each configuration. On the Table 2 we can determine that at a given price, the best performance will provide the configuration 4, that consists of 25 type 4 nodes.

Example 2. Let the organization have purchased the computing cluster for database application in 2009. This cluster consists of 20 type 3 nodes. Assume that by 2011 the size of database application has increased by 1.5 times and it became necessary to expand the system. We have to determine how many type 3 nodes should be purchased to increase the system performance by 1.5 times. The current system processes the transactions for 222700 arbitrary units of time, according to Fig. 9. Respectively, the system with 1.5 times higher performance should process equal mix of transactions for 148467 arbitrary units of time. Using Fig. 9 we can determine that the system that consists of 31 type 3 nodes will provide the specified performance. Thus, it is necessary to purchase 11 additional type 3 nodes for €89,375.

Example 3. Let the organization have purchased the computing cluster for database application in 2009. This cluster consists of 26 type 2 nodes. Assume that by 2011 the size of database application has increased by 1.5 times and we know that we know that the same growth dynamics of database size will continue in the next two years. We have to make the first step of system upgrade in such a way that total cost of the two steps of extension (in 2011 and 2013) was minimal.

Table 3. Performance requirements

Step	Year	Time (in arbitrary units)
1	2009	346722
2	2011	231148
3	2013	154099

Table 4. Variants of configuration

Node types	Number of nodes		
	Year 2009	Year 2011	Year 2013
2	26	39	57
4	–	10	15

The requirements for the performance of computing cluster in 2009, 2011 and 2013 are shown in Table 3. These requirements are based on dependencies in Fig. 9. The variants of allowable configurations that will provide the required performance are shown in Table 4.

We consider two variants of the first upgrade step (year 2011):

A.1) expansion of the current system by adding the necessary number of type 2 nodes;

B.1) replacement of the current cluster to the new with type 4 nodes.

Using Table 4 we can see that in case of variant A.1 we have to purchase 13 type 2 nodes. Using Table 1 we calculate that the cost of variant A.1 is €96,850. Analogically we get that the

* Supported by the Russian Foundation for Basic Research under Grant 12-07-00443-a and by the Federal Program 07.514.11.4036.

variant B.1 is the purchasing of 10 type 4 nodes, which cost €98,750.

In line with what variant we choose, we may get two variants on the second step of upgrade in 2013:

A.2) expansion of the system by adding the necessary number of type 2 nodes;

B.2) expansion of the system by adding the necessary number of type 4 nodes.

Using the similar calculations we get that variant A.2 is to purchase 18 type 2 nodes, which cost €13,2750, and variant B.2 is to purchase 5 type 4 nodes, which cost €43,250.

Table 5. Upgrade cost

Upgrade variant	Step		In total (€)
	1	2	
A	96,850	132,750	229,600
B	98,750	43,250	142,000

The total cost of two types of upgrade is shown in Table 5. We can see that variant A is more profitable on the first step, but after step 2 it is more expensive than variant B for €87,600.

4 Conclusion

In this paper, we introduced the new computational model of the hierarchical database multiprocessor architecture. This model is called *DMM* (*Database Multiprocessor Model*) and designed for *OLTP* workload. It allows us to simulate and analyze an arbitrary multiprocessors configuration for database application.

The *Database Multiprocessor Simulator* is developed on the basis of *DMM*. It allows to simulate and investigate various multiprocessor configuration for *OLTP* workload. The experiments for search for optimal hardware architectures of parallel database systems were performed with the simulator.

The main directions of future work are the following. First, it is interesting to use *DMM* for simulating parallel database systems' processing on the petascale multiprocessor systems based on NVIDIA Tesla and Intel Many Integrated Core (MIC) accelerators [11]. Second, we plan to apply *DMM* model for simulating the parallel database systems processing in grid. Third, we suppose to extend the *DMM* model for OLAP applications.

References

[1] Agrawal R., Carey M.J., Livny M. Concurrency Control Performance Modeling: Alternatives and Implications // ACM Transactions on Database Systems. Vol. 12, No. 4. 1987. P. 609-654.

[2] Becla J., Wang D.L. Lessons Learned from Managing a Petabyte // CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings URL: http://www.cidrdb.org/cidr2005 (дата обращения: 08.11.2009). 2005. P. 70-83.

[3] Bhide A., Stonebraker M. Performance Issues in High Performance Transaction Processing Architectures // Proceedings of the 2nd International Workshop on High Performance Transaction Systems, September 1987, Asilomar. Springer-Verlag, 1989. Vol. 359. P. 277-299.

[4] Bhide A., Stonebraker M. A Performance Comparison of Two Architectures for Fast Transaction Processing // Proceedings of the Fourth International Conference on Data Engineering, February 1-5, 1988, Los Angeles, California, USA. IEEE Computer Society, 1988. P. 536-545.

[5] Bhide A. An Analysis of Three Transaction Processing Architectures // Fourteenth International Conference on Very Large Data Bases (VLDB'88), August 29 - September 1, 1988, Los Angeles, California, USA, Proceedings. Morgan Kaufmann, 1988. P. 339-350.

[6] Carey M., Stonebraker M. The Performance of Concurrency Control Algorithms for Database Management Systems // The 10th VLDB Conference, August 1984, Singapore, Proceedings. Morgan Kaufmann, 1984. P. 107-118.

[7] Carey M.J., Livny M. Distributed Concurrency Control Performance: A study of Algorithms, Distribution, and Replication // The VLDB Conference, Los Angeles, California, Proceedings. Morgan Kaufmann,1988. P. 13-25.

[8] Carey M.J., Livny M. Parallelism and Concurrency Control Performance in Distributed Database Machines. Proceedings of the 1989 ACM SIGMOD Inter. Conf. on the Management of Data, Portland, Oregon, Vol. 18, No. 2. ACM, 1989. P. 122-133.

[9] DeWitt D., Ghandeharizadeh S., Schneider D., Bricker A., et al. The gamma database machine project // IEEE Trans. Knowledge Data Eng. 1990. Vol. 2, No. 1. P. 44-62.

[10] Hsiao H.I., DeWitt D.J. A Performance Study of Three High Availability Data Replication Strategies // Distributed and Parallel Databases. 1993. Vol. 1, No. 1. P. 53-80.

[11] Hughes C.J, Changkyu K., Yen-Kuang C. Performance and Energy Implications of Many-Core Caches for Throughput Computing // IEEE Micro. 2010. Vol. 3, No 6. P. 25-35.

[12] Livny M. DeNet User's Guide, Version 1.0. Comp. Sci. Dept., Univ. of Wisconsin, Madison, 1988.

[13] Sokolinsky L.B. Survey of Architectures of Parallel Database Systems // Programming and Computer Software. 2004. Vol. 30. No. 6. P. 337-346.

[14] Xu Y., Dandamudi S.P. Performance Evaluation of a Two-Level Hierarchical Parallel Database System // Proceedings of the Int. Conf. Computers and Their Applications, Tempe, Arizona. 1997. P. 242-247.

* Supported by the Russian Foundation for Basic Research under Grant 12-07-00443-a and by the Federal Program 07.514.11.4036.

An Adaptive Storage and Retrieval Mechanism to Reduce Response-Time in High Performance Computing Clusters

Amir Saman Memaripour, Ehsan Mousavi Khaneghah, Seyedeh Leili Mirtaheri, and Mohsen Sharifi
School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

Abstract - *Ever-increasing growth of high performance computing applications requires employment of novel methods in all aspects of computing systems. The response time of file storage and retrieval operations is one of the most important factors of storage systems and improving that will result higher computational power. Consequently, breathtaking efforts have been done and various file systems with different architectures have been proposed. Most of them are not aware of clusters' execution state and do not consider variety of I/O operations' response time on machines with different storage media, network traffic, and processing load. In this paper, we have proposed a mechanism to store and retrieve files with respect to the execution state of storage nodes and network topology of the cluster. Finally, the proposed architecture has been implemented and evaluated using Hadoop distributed file system.*

Keywords: File System, Response Time, Adaptive Storage and Retrieval, HPC

1 Introduction

Exponential growth of the volume of stored data has made conventional methods of storage and retrieval inefficient which require new ways with higher performance to deal with [1]. Performed estimations show that the size of electronic data was about 1.8 zeta bytes in 2011 that illustrates the need to store tremendous amounts of data [2]. Furthermore, improving the performance of these operations in various aspects such as scalability and reliability must be considered [3]. In this way, three different approaches including improving the performance of each computational element, revising the algorithms, and making use of a series of ordinary computing elements could be followed [4].

In the field of HPC computing, improving overall performance is mainly focused on reducing response time and increasing processing power of the system [4]. In order to fit in with the requirements of these environments and avoid becoming the bottleneck, file system designers have used several techniques to reduce the response time [5] [6]. Numerous file systems, such as HDFS and PVFS, have been proposed to cope with these issues and used novel algorithms and load balancing techniques to improve the overall system performance with the aid of distributed computing [7] [8].

Heterogeneous storage clusters are made of various computational elements with different performance levels. This diversity leads to different response times of computational nodes to a single request. Moreover, the response time of each node depends on other parameters such as processor load, network traffic, and remaining storage space changing during time and will affect the performance of the node [9]. Network topology and the distance between data consumer and data provider is another important parameter that can affect the performance of I/O operations. In many HPC clusters, applications and their respective data are placed as closest as possible and data access is performed using a high-bandwidth and low-latency communication network. However, communication cost between two adjacent nodes is far less than two nodes which are located in different parts of the communication network, i.e. two different racks [7].

In these clusters, data replication is performed in order to improve reliability and response time [7]. If the number and the place of replicas are determined with respect to the execution state of computational nodes, the response time and scalability of the cluster will be improved. In addition, the load will be distributed among the nodes and data will not aggregate in particular parts of the cluster. In this manner, an adaptive mechanism that replicates and distributes replicas in the cluster according to the execution state of cluster's nodes has been proposed in this paper.

2 Related works

Various usages of file systems are possible in computing systems. For example, resources could be represented, accessed, and managed by files in distributed computing systems [10]. Moreover, files could be used to store, retrieve, name, structure, protect, and manage user and system data. Our proposed mechanism considers file systems as the module which is responsible for the mentioned responsibilities in the second definition.

Remarkable efforts have been done in the field of file systems which can be categorized in three classes including centralized, decentralized, and distributed file systems. Traditional file systems like FAT belong to the first class and are inefficient for high performance applications [11]. Extremely low scalability and the limitations of storage media like disks' seek time lead to the appearance of the other two classes. Decentralized file systems were mainly proposed to increase the number of master nodes in file systems and avoid consequences of having a single point of failure. NFS is one of the centralized file systems that a decentralized version of it has been proposed to increase its scalability [12].

Computation requirements of today's applications need a level of scalability which is far from what decentralized approaches can reach. As a result, numerous file systems with distributed architectures have been presented including HDFS and GFS which are used to maintain organizational data at

Yahoo and Google respectively. These two are also good examples of the successful experiences in increasing scalability [7] [13].

PVFS was introduced for parallel storage and retrieval of files in Linux clusters and parallel execution of applications which were previously performed using parallel machines. This file system was designed to support different file system interfaces like POSIX, facilitate installation and use of the file system, and provide features to store and retrieve files in a parallel and concurrent manner by several processes. Moreover, scalability and robustness were among other goals in the design process of this file system [8]. PVFS has been developed as a base for research and development in the field of parallel file systems in Linux [1] [5] [14]. Furthermore, a branch of PVFS called OrangeFS has been developed to consider issues like metadata operations, blocks replication, and access control in more depth [15].

In 2003, Google presented GFS which had been used before that time by Google's applications and designed according to their storage requirements. In addition to different aspects of performance which were addressed in other distributed file systems, GFS has paid special attention to efficient execution of its applications [13]. GFS do not support POSIX standard, has larger basic storage blocks comparing to traditional file systems, and is mainly focused on the optimization of append operations because of the nature of its applications [13] [16]. Moreover, using commodity and cheap hardware in the implementation process of GFS made hardware failure such as network devices and storage media a probable event rather than unexpected. In order to prevent the consequences of these failures, several innovative methods have been considered in the design of GFS.

Ceph is another distributed file system which is proposed to provide a scalable, reliable, and high performance storage cluster [17]. It separates the management of data and metadata and supports more than 250,000 metadata operations in every second by using a pseudo-random distribution function, called CRUSH, instead of allocation tables [18]. Moreover, it distributes the replication, fault detection, and recovery operations among storage nodes, where data and metadata are stored. Ceph is composed of three major parts including client, storage cluster, and metadata cluster. It also provides a near-POSIX file system interface which is a remarkable feature for a distributed file system.

Another project called HDFS is currently in progress in Yahoo that is similar to GFS in various aspects and is published as an open-source file system [7]. HDFS performs as a reliable and high performance storage cluster and used several concepts proposed in GFS. More detailed information about this file system is available in section 3, where implementation platform of our proposed mechanism has been introduced.

Common file systems in this area have not effectively considered execution state of the cluster and computational nodes. In addition, heterogeneity of storage media is another parameter that affects response time and has not properly addressed in these file systems. Our proposed mechanism, discussed in more detail in section 4, has considered these parameters and introduced a method to improve the response time of storage and retrieval operations.

3 Implementation platform

In order to implement and evaluate our proposed mechanism, we have used Hadoop Distributed File System as our implementation platform. HDFS is designed to reliably store large data sets and stream these data to user applications at high bandwidth. Distributing stored data and computations among several processing and storage nodes in HDFS leads to a scalable and economic cluster that can grow to any size on demand [7]. This file system has being used by Yahoo to manage 25 petabytes of application data and has been successful in this way [2]. We have stated our motivations to use it as our implementation platform in section 3.2 and after a brief introduction to HDFS.

3.1 Overview

Every HDFS cluster at least consists of a NameNode and one or more DataNodes. The cluster can grow to any size by adding more DataNodes to it [19]. Moreover, other nodes such as BackupNode and CheckpointNode can join the cluster to improve its reliability [7]. User applications, usually executed on DataNodes, are other pieces of the architecture and interact with HDFS using a programming library. This file system does not support POSIX standard and its interfaces are provided via HDFS client which is a code library [2].

In this architecture, NameNode maintains and manages namespace. Moreover, it manages DataNodes and replicas of each data block. The namespace is a hierarchical structure of directories and files which are represented by inodes. Each file is divided into large data blocks which could be replicated independently and stored on various DataNodes. Besides maintaining the file system namespace, NameNode stores required information to perform the mapping between data blocks and DataNodes [7].

DataNodes are responsible for maintaining the replicas of data blocks in their local storage media and store two files for each replica. One of the files is used to store the data and the other maintains block's metadata. Each DataNode performs a handshake with the NameNode at the startup. During this operation, namespace identifier and software version of the NameNode are compared with respective values of the DataNode. Consequently, the consistency of the file system will be preserved [7]. DataNodes send block reports to the NameNode after joining the cluster and periodically to inform it about their hosted replicas. Moreover, the NameNode receives another kind of control messages from DataNodes that indicates their presence in the cluster and accessibility to their hosted replicas [2].

3.2 Motivations in using HDFS

Availability of user manuals and books besides its open source implementation made it feasible to use HDFS as the implementation platform [2]. In addition, it is widely used in extraordinary computing environments and has the potential to be applied on high performance computing clusters.

Performing the replication under the supervision of the NameNode provides required information for adaptive distribution of replicas over the cluster and makes it a good choice to run the proposed mechanism in this node. Moreover, the execution state of computational nodes can be send along with control messages which are periodically sent by DataNodes to the NameNode.

The mechanism used by HDFS to provide user applications with the block replicas list can be changed to consider the execution state of DataNodes and perform adaptively. HDFS has also provided an interface to consider the network distance between data providers and consumers in order to sort hosts in the replica lists [7]. Furthermore, this file system is designed to use commodity hardware that makes evaluation feasible. HDFS is implemented using JAVA programming language that facilitates source code modification and makes it possible to run it on a wide variety of operating systems [2]. Considering these facts, we have chosen HDFS as our implementation platform in order to evaluate our proposed mechanism.

4 Proposed mechanism

The goal of our proposed mechanism is to consider the execution state of DataNodes and the topology of communication network in storing and retrieving data blocks. Adapting file system operations to the execution state of the cluster will improve the overall performance and distribute the I/O load across the storage cluster. Moreover, selecting least loaded nodes to host replicas can improve the response time of I/O operations.

The execution state of DataNodes including remaining storage space, available network bandwidth, and I/O speed of storage media and where it is placed in the communication network are considered with regard to selecting an appropriate node. Besides adapting I/O operations to execution state of nodes, our proposed mechanism introduces a method to automatically detect the network topology that will help data blocks distribution in the storage cluster.

4.1 Architecture

Adapting reading and writing data blocks to execution state of DataNodes and the network topology is performed by three modules. One of these modules concerned about the network topology and is responsible for maintaining the Network Tree which is used by the other modules. Construction of the Network Tree is performed according to relative distance between computational nodes and communication switches. For instance, direct connection of two nodes to a switch means both of them belong to a same rack. In the same way, relative distance between different racks is estimated using the communication delay between their computational nodes. Section 4.3 contains detailed information about this module.

Adaptive generation of the list of DataNodes hosting replicas of a data block is performed by another module which is described in more detail in section 4.2. This module estimates transfer time of the replica according to the last known load of the DataNodes and the network distance between each DataNode and the client.

The other module of the proposed mechanism selects destination DataNodes to host replicas of each data block. In order to select more appropriate nodes, a priority value is assigned to each DataNode and is continually being updated based on its workload. Using these priority values and the network distance between DataNodes and the client, this module will choose the best node to host the replicas. More information about this module has been stated in section 4.4.

4.2 Adaptive Generation of Replica-Hosts List

Once the NameNode receives a request from a client for reading a data block, it provides the client with a list of DataNodes hosting replicas of the requested data block. HDFS only considers the network distance between the client and replica hosts in construction of this list and ignores the execution state of DataNodes [7]. Our proposed mechanism estimates the time that C_i will spend for reading B_k from D_j and orders the replica-hosts list by these estimated values. As a result, the overhead of read operations will distribute across DataNodes that will improve the overall response time of the cluster. Moreover, some required computations of this mechanism are performed by DataNodes that avoids addition of a considerable overhead to the NameNode.

$$EstReadTime(C_i, D_j, B_k) = size(B_k) \times ReadLatency(D_j) + NetDist(C_i, D_j) \quad (1)$$

Equation 1 is used to estimate the time required for reading a data block called B_k from D_j by the client that is running on C_i in seconds. The value of $size(B_k)$ is stored on the NameNode and can be fetched by one memory access. Furthermore, required computations for $ReadLatency(D_j)$, the amount of time in seconds to read one megabyte of data from D_j, is performed by the respective DataNode and the result is sent to the NameNode via heartbeat messages. Equation 2 shows how this value is calculated. The network distance between C_i and D_j is estimated based on the network topology using $NetDist(C_i, D_j)$. Section 4.3 contains detailed information about this function and the way it detects the network topology.

$$ReadLatency(D_j) = \frac{1}{ReadSpeed(D_j)} + \frac{1}{(1-NetUt(D_j)) \times netSpeed(D_j)} \quad (2)$$

In order to estimate $ReadLatency(D_j)$ in Eq. 2, we need to be aware of the execution state of D_j including the transfer speed and the utilization of its NIC besides the I/O speed of its storage media. The momentary I/O speed of the storage media can be estimated based on the current data streams of the DataNode. In order to rate the maximum I/O speed of the storage media, a shell script will be executed at each startup of the DataNode. Using this information, Eq. 3 estimates $ReadSpeed(D_j)$ in megabytes by calculating a weighted-average of the capacity of all local storages in D_j. Similarly, network transfer speed is evaluated by multiplication of network utilization and total transfer speed of the NIC in MB/s. In this manner, the total transfer speed of each network

interface is determined by parsing the respective output of *ethtool* in Linux. Finally, the addition of inverse of these values represents the time required for reading one megabyte of data from D_j by any other node in the cluster.

$$ReadSpeed(D_j) = \frac{\sum_{i=1}^{n} UsedSpace(S_i) \times (TotalSpeed(S_i) - UsedSpeed(S_i))}{\sum_{i=1}^{n} UsedSpace(S_i)} \forall S_i \in D_j \quad (3)$$

4.3 Dynamic network topology detection

In order to estimate the network distance between DataNodes, an n-ary tree called Network Tree is used. The interior nodes of this tree represent communication switches, while its leaves contain DataNodes of each rack. The branching factor of this tree is the maximum number of ports of all communication switches and is denoted by *BF*. As a result, the height of this tree can be calculated by $h=\lceil \log_{BF} n \rceil$ for some positive *n* that reflects the number of racks in the cluster. Moreover, every leaf of the Network Tree is assigned with a unique identifier called RackID which is used for indexing all the DataNodes belonging to a same rack [20].

Once a DataNode joins the cluster, it is considered to be a member of a new rack and is added to the Network Tree as the immediate successor of the root. After that, the Network Tree and the indexed list of DataNodes will be sent to the newly joined DataNode to find its place in the network. Fig. 1 contains the pseudo-code of the search algorithm a DataNode follows to find its place in the network along with the structure of the Network Tree. The result of the performed search in the newly joined DataNode will be sent to the NameNode in order to update the Network Tree.

```
INPUT: N_c = current DataNode
INPUT: p = the root of NetworkTree
OUTPUT: R_id = null
while (R_id == null) do
    last_RTT = 0
    for each node_i in childs(p)
        N_D = select one of the descendent DataNodes of node_i in a random manner
        new_RTT = eval_RTT(N_c, N_D)
        if (new_RTT < last_RTT) {
        if (childs(node_i) == null) R_id = node_i
        p = switch_i
        break
        }
        last_RTT = new_RTT
```

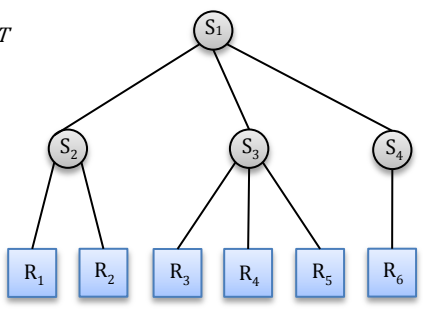


Figure 1: The structure of Network Tree and pseudo-code of its search algorithm

By identifying the nearest common predecessor of two DataNodes in the Network Tree and counting the number of interior nodes in their communication path, the NameNode can estimate the communication latency between every two DataNodes in the cluster. The NameNode assumes a constant communication delay for each switch in the communication path in order to estimate the total communication latency. The running time of this algorithm is $O(\log^{BF}{}_{n})$. Considering the large values of *BF*, its overhead could not be remarkable. As an example, assuming *n*=350 and *BF*=16, the height of the Network Tree would be 3 that supports our point.

4.4 Adaptive selection of replication hosts

Once a new data block is created or an existing one is going to be replicated, the NameNode should select some DataNodes to host the newly created replicas. This selection can be performed based on various considerations such as reducing replication time, improving data distribution, reducing the distance between data provider and consumer, and hosting replicas in the nearest racks. Our proposed mechanism assumes that reader processes are uniformly distributed across the cluster, so it tries to select the more appropriate DataNodes from the nearest racks to the data source. In addition, it obeys the replication rules of HDFS and stores a data block in a rack that contains at most one of its replicas. At the final step of selection, qualified DataNodes are ordered by their remaining storage space and available network bandwidth using Eq. 4. After that, the most proper nodes will be chosen to host the recently created replicas.

$$NodeRate(D_j) = Min\left(\alpha \times \frac{FreeSpace(D_j)}{maxStorageSize}, \beta \times \frac{AvailBW(D_j)}{maxNetBW}\right) \forall D_j \in Cluster \quad (4)$$

The selection priority of every DataNode is determined by comparing its free storage space and available network bandwidth with the total storage space and maximum network bandwidth of the cluster. *FreeSpace*(D_j), *AvailBW*(D_j), *maxStorageSize* and *maxNetBW* are calculated using the information which is being periodically received from DataNodes. Moreover, the effect of free storage media and available network bandwidth on *NodeRate*(D_j) can be adjusted using α and β constants. Here, we have used α=1 and β=1 for the sake of simplicity.

The respective statistics and information of all DataNodes of each rack is stored in a max-heap according to their selection priority value [20]. These max-heaps are referenced by the leaves of the Network Tree and are accessible in this way. As a result, selection of the most proper DataNode of a rack can be performed in O(1). Fig. 2 shows the pseudo-code of the algorithm for selecting the DataNodes to host the recently created replicas of the data block hosted by *Di*.

Once a new data block is created in a DataNode, the first replica is stored in the same node. The other two DataNodes to store replicas are selected using the algorithm shown in Fig. 2 and a pipeline will be organized by HDFS to transfer replicas to the selected hosts. By default, this algorithm stores the replicas in the nearest racks to the data source. However, other policies to select replication hosts could be used in order

to improve reliability of the cluster. For example, storing a replica in another part of the cluster can keep the data block accessible on the event of communication switch failures.

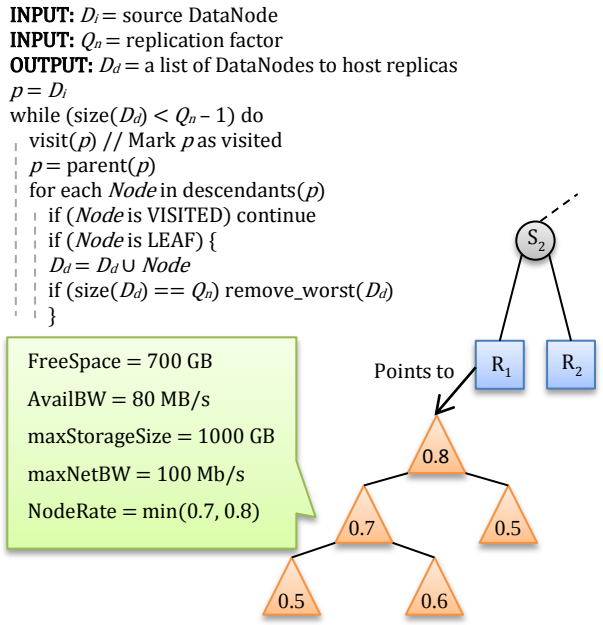


Figure 2: The structure of leafs of Network Tree and the psuedo-code of host selection algorithm

Replication of an existing data block is performed in a similar way. In order to find the proper DataNode to host the new replica, every DataNode that hosts one of the replicas of the data block selects a host using an algorithm similar to the one shown in Fig. 2 with Q_n=1. Next, the proper host for the new replica will be chosen from the selected hosts in the previous step and replication will be performed by the nearest DataNode to it. HDFS performs the replication whenever one of the replicas of a data block becomes corrupted, so the execution rate of this mechanism cannot be remarkable. Moreover, this mechanism can be used to adapt the number of replicas with a dynamic replication factor in storage clusters.

5 Evaluation

We have measured the performance of the proposed mechanism using three distinct criterions. The first two evaluations were performed using HDFS benchmarks in order to demonstrate the improvements in I/O performance of DataNodes and response time of the NameNode. The third one was performed to verify the low computational cost, network traffic and execution time of the algorithm proposed for Network Tree construction. According to the evaluation results, the proposed mechanism can improve the response time of file storage and retrieval operations in heterogeneous clusters with long-term I/O operations.

5.1 I/O performance

In order to evaluate the I/O throughput of HDFS in presence and absence of our proposed mechanism, we have used TestDFSIO benchmark, one of the benchmark packages of hadoop-test-1.0.1. In this manner, a cluster of one master node and three slave nodes is used. Hardware configuration and OS specification of these systems is shown in Table 1.

Table 1: Hardware and OS specification of cluster nodes

	OS	CPU	RAM	Disk (RPM)
Master	Mac 10.6.8	Intel Core 2 Due 2.8	4 GB	5400
Slave	Ubuntu 10.04	Intel Pentium IV 3 GHz	2 GB	5400

TestDFSIO is a MapReduce job and proceeds in two distinct phases. First, it creates some map and reduce tasks to write a number of files which is specified as an input parameter. After that, some other tasks are created to estimate the read performance of the cluster by reading specified number of files. In each phase, the benchmark prints some statistical information about the I/O operation performance of the cluster. In order to examine the effect of our proposed mechanism on HDFS I/O performance, TestDFSIO is executed in the presence and absence of the mechanism using various input parameters and the results are shown in Table 2.

Considering the presented evaluation results, our proposed mechanism is more beneficial for storage clusters with long-term I/O operations. As a result, increasing the size of files will magnify the effect of adaptive selection algorithms on the performance of storage clusters.

Table 2: Evaluated I/O throughput using TestDFSIO

Number of files		10	100	10	100
File size (MB)		100	100	1000	1000
Read (MB/s)	HDFS	7.006	7.421	8.983	9.603
	HDFS + AM	7.043	7.492	9.217	10.039
Write (MB/s)	HDFS	4.893	5.161	6.037	6.958
	HDFS + ASM	4.923	5.202	7.686	7.278

5.2 Cluster scalability

Execution overhead of the proposed mechanism can degrade the number of metadata operations that the NameNode is capable of performing in every second. Once the NameNode reaches its resource limits as a result of an extraordinary number of metadata operations, presence of the proposed mechanism can increase response time of metadata operations and decrease overall performance of the cluster. For investigating this issue, one of the benchmark packages of hadoop-test-1.0.1 called NNBench is used. This benchmark performs in four distinct phases including *create_write*, *open_read*, *rename*, and *delete*. Moreover, NNBench supports a wide variety of input parameters including *sizeOfBlocks* and *numberOfFiles*. Fig. 3 shows the execution results of NNBench in the presence and absence of our proposed mechanism.

As it can be seen in Fig. 3, execution overhead of the proposed mechanism can have a slight impact on the NameNode performance only when the NameNode is overwhelmed with user requests. Indeed, this performance degradation is a direct outcome of centralize nature of the

NameNode in Hadoop clusters and can be avoided on the first sight by downscaling the cluster or improving hardware specification of the NameNode. However, using the proposed mechanism looks economical and effective with regards to how much it improves the system performance by spending a slight amount of processing power and memory space.

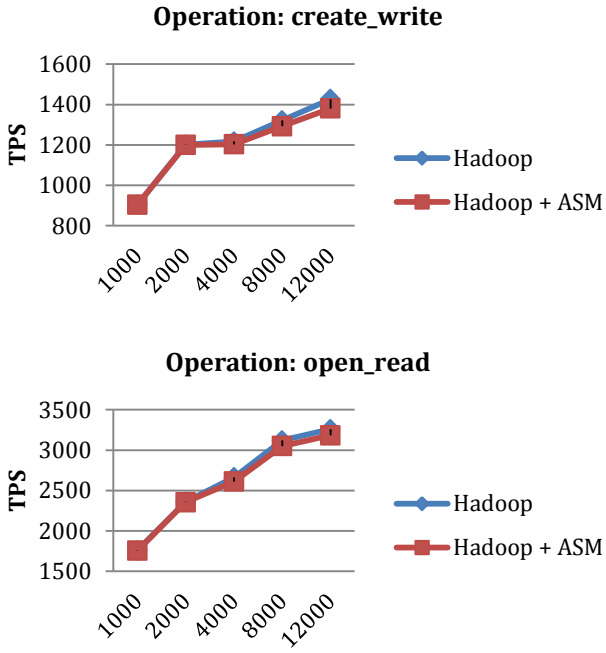


Figure 3: NameNode TPS evaluation using NNBench

5.3 Construction cost of Network-Tree

Construction of Network Tree is performed using ICMP network messages and the Ping utility. To evaluate the performance of our proposed method, we have examined the total network traffic and construction time of this method. Total network traffic of each run is monitored using Wireshark [21]. Searching the Network Tree is a one-time operation and every DataNode have to perform it just after joining the cluster. D-Link switches with a total bandwidth of 100 Mbps have been used to from the communication network and connect computational nodes running a Linux distribution. The height of the Network Tree is two that demonstrates a two level network topology including the core switch and rack switches which are directly connected to the core switch. Fig. 4 shows our experimental results, in which total network traffic and the execution time are in KB and seconds respectively.

The execution time of this algorithm, which is used for searching the Network Tree, directly depends on the branching factor and the height of the tree. Height of the tree does not vary much and even in extra-large clusters, it can hardly ever reach a value more than 3. For instance, using 16-port switches in building a communication tree with the height of 3 will lead to a cluster of size 16^3. The largest operational Hadoop cluster at Yahoo contains 3500 DataNodes which is far less than 4096. Moreover, the branching factor does not exceed 16 most of the times. In most of the clusters, network switches with more than 16 ports are used as rack switches that have no effect on the branching factor of the Network Tree. Considering how this method can facilitate joining of new DataNodes to the cluster, it can be inferred that its execution time and network traffic are reasonable.

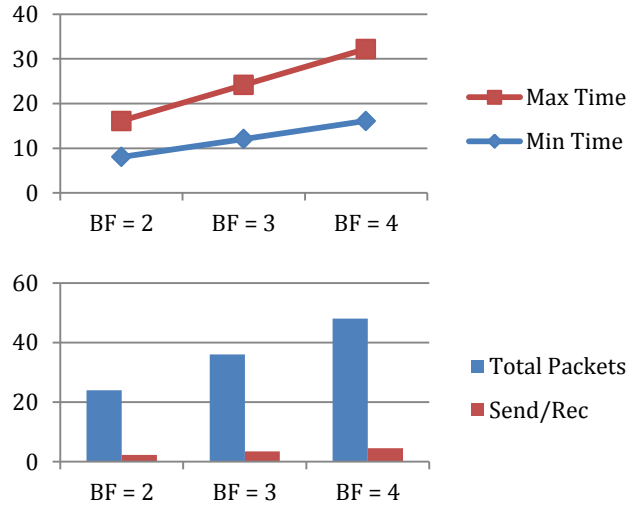


Figure 4: Evaluation results of the tree construction algorithm

6 Further works

Besides the proposed mechanism, distributed file systems can adapt with the execution state of the cluster in some more ways. Using dynamic replication factors and popularity domains for files are two of these ways that can benefit the storage cluster in both reliability and response time.

Considering the concept of files' popularity which is widely used in P2P file sharing systems, the replication policy can be improved. Thus, each file will be replicated in every neighborhood that contains more reader or writer nodes. In this way, a popularity domain will be assigned to each file containing the DataNodes that will probably read or write from/to it in the future. This mechanism can be used along with the task distribution mechanism of Hadoop which executes every task near to its required data. The cooperation of these two can lead to better distribution of load in Hadoop clusters. As a result, the behavior of file system users will affect the replication of data blocks and the file system will adapt with users' activity.

Besides being used for determining proper replication factor of each file, the concept of popularity domains is beneficial for improving reliability of the NameNode in Hadoop clusters. As each popularity domain normally contains the list of DataNodes sharing the respective file, the global namespace can be divided into many popularity domains and distributed across several NameNodes. Consequently, reliability of the central mechanism which is currently in use in Hadoop clusters to maintain file system metadata will be improved by increasing the number of metadata servers. This mechanism follows the same idea as namespaces in Plan 9 and uses popularity domains in exchange of per-process namespaces. Reducing the computational overhead of the NameNode will directly improve the response time of storage and retrieval

operations of files. Indeed, this mechanism can improve both reliability and response time of distributed file systems.

7 Conclusion

Our proposed mechanism considers the execution state of the storage cluster in performing data blocks replication. Various aspects of the cluster such as network topology and different state variables of computational nodes like network traffic and remaining storage space are considered to find the proper host for storing replicas. As the first step in the replication process, the NameNode specifies the required number of replicas for each data block. Next, the replication roadmap is designed based on the performed calculations and the selected host will store the newly created replica. The host selection algorithm will choose the proper nodes to store the new replica by looking at their execution state, available resources, and distance to the source of data. As a result, the load of storage and retrieval operations will be uniformly distributed across the cluster.

A similar method is used in order to find proper replica hosts to provide read operations with requested data blocks. In this manner, our proposed mechanism estimates the cost of reading a data block from every host and uses the results to generate an ordered list of hosts. Some execution variables of storage nodes including available network bandwidth, disk read speed, and current data streams are considered in evaluating the cost of reading every data block. The network distance between the replica host and reader node is also considered in this selection algorithm.

HDFS has been used as our implementation platform. Several facts like easy to use interfaces and open source implementation of HDFS lead to this decision. Performed evaluations show that our proposed mechanism improves the response time of file storage and retrieval operations in heterogeneous clusters running long-term I/O operations. Furthermore, its computational overhead will not have a remarkable effect on cluster scalability.

8 References

[1] R. Ross, D. Nurmi, A. Cheng, and M. Zingale, "A Case Study in Application I/O on Linux Clusters," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 2001, pp. 11-11.

[2] T. White, *Hadoop: The Definitive Guide, 2nd Edition*. O'Reilly Media / Yahoo Press, 2010.

[3] B. Witworth, J. Fjermestad, and E. Mahinda, "The Web of System Performance," *Communications of the ACM*, vol. 49, no. 5, pp. 93-99, May 2006.

[4] R. Buyya, *High Performance Cluster Computing*. Prentice Hall, 1999.

[5] W. Yu, S. Liang, and D. K. Panda, "High Performance Support of Parallel Virtual File System (PVFS2) over Quadrics," in *Proceedings of the 19th Annual International Conference on Supercomputing*, New York, NY, USA, 2005, pp. 323-331.

[6] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-Generation Data Management for Large Scale Infrastructures," *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, p. 169–184, Feb. 2011.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *26th Symposium on Mass Storage Systems and Technologies (MSST)*, Incline Village, NV, 2010, pp. 1-10.

[8] P. H. Carns, W. B. Ligon, and R. T. Robert B. Ross, "PVFS: A Parallel File System for Linux Clusters," in *Proceedings of the 4th Annual Linux Showcase*, Berkeley, CA, USA, 2000, pp. 28-28.

[9] J. Montes, B. Nicolae, G. Antoniu, A. Sánchez, and M. S. Pérez, "Using Global Behavior Modeling to Improve QoS in Cloud Data Storage Services," in *Second International Conference on Cloud Computing Technology and Science (CloudCom)*, Indianapolis, IN, 2010, pp. 304-311.

[10]D. Presotoo, R. Pike, K. Thompson, and H. Trickey, "Plan 9, A Distributed System," AT&T Bell Laboratories, 1991.

[11]W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs," in *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Santa Clara, California, United States, 2000, pp. 34-43.

[12]T. E. Anderson, et al., "Serverless Network File Systems," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 109-126, Dec. 1995.

[13]S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29-43, Oct. 2003.

[14]A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan, "Integrating Parallel File Systems with Object-Based Storage Devices," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 2007, pp. 1-10.

[15]OrangeFS. (2011, Dec.) Orange File System. [Online]. http://www.orangefs.org/

[16]J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, Jan. 2008.

[17]S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *7th Symposium on Operating Systems Design and Implementation*, Washington, 2006, pp. 307-320.

[18]S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 2006, pp. 31-31.

[19]J. Venner, *Pro Hadoop*. Apress, 2009.

[20]T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[21]Wireshark Foundation. (2012, Feb.) Wireshark. [Online]. http://www.wireshark.org/

A Learning Algorithm of Threshold Value on the Automatic Detection of SQL Injection Attack

Daiki Koizumi*, Takeshi Matsuda*, Michio Sonoda*, and Shigeichi Hirasawa*
* Faculty of Information Technology and Business, Cyber University, Tokyo, Japan

Abstract— *The SQL injection attack causes very serious problem to web applications that involve database including personal data. To detect the SQL injection attack, the parsing and the blacklist built based on the known attacks have been widely used. Those approaches, however, have some problems in terms of the size of list or calculation costs as the number of attacks increases. For these problems, the authors have previously proposed a simple automatic algorithm to detect SQL injection attack. This algorithm requires to calculate the ratio of suspicious characters contained in an input sequence. This rate is compared with a known real-valued threshold. This paper proposes the learning algorithm to choose the real-valued threshold from training data sets. Furthermore, some criteria will be considered and their performances will also be examined.*

Keywords: SQL Injection Attack, Automatic Detection, Web Application, Learning Algorithm, Pattern Recognition

1. Introduction

The SQL injection attack was firstly reported in 1999 [2]. It consists of insertion or "injection" of a SQL query via the input data from the client to the application [3]. It causes very serious problem to web applications that involve database including personal data. To detect the SQL injection attack, the parsing and the blacklist built based on the existed attack have been widely used. However, the conventional methods based on blacklist built approaches have confronted at least two problems. The first is the huge cost of updating large blacklist built, the second is the calculation cost for automatic attack detection with large blacklist built.

For these two problems, the authors have already proposed the online algorithm for automatic detection of SQL injection attack [4]. This algorithm requires to calculate the contained rate of suspicious characters with input sequence. This rate is compared with a known real-valued threshold. In the authors' previous research [4], this threshold was empirically determined as a known constant. However, in general, this threshold is an unknown value and should be learned from a given training set of data. This paper proposes a learning algorithm to choose a real-valued threshold. Furthermore, some criteria will considered and their performances will also be examined.

The rest of this paper is organized as the followings. Section 2 gives some definitions and automatic detection algorithm of SQL injection attack which was previously proposed where its threshold value is defined as a known constant. Section 3 proposes the learning algorithm of the threshold value with some formulations. Section 4 shows some leaning examples with artificial data. Section 5 gives their discussions. Finally, Section 6 concludes this paper.

2. The Automatic Detection Algorithm of SQL Injection Attack [4]

2.1 Preliminary

Suppose that l is an input sequence through web application to the SQL database. Note that each input l has a label either attack or normal. The purpose of automatic SQL injection attack detection is to estimate the label of l correctly. Our proposed algorithm requires some known finite characters $s_i, s_{ii}, s_{iii}, \cdots$. These are suspicious characters contained in SQL injection attack inputs such as space, semicolon, single-quotation, left and right round brackets, and so on. Furthermore, let S_k, $k = 1, 2, \cdots, m$ be power sets of known characters $s_i, s_{ii}, s_{iii}, \cdots$ where they are defined as sets of characters except for the empty set. If s_i and s_{ii} are defined as space and semicolon, then three power sets of known characters can be $S_1 = \{s_i\}$, $S_2 = \{s_{ii}\}$, and $S_3 = \{s_i, s_{ii}\}$.

2.2 The Automatic Detection Algorithm

With the above definitions, the automatic detection algorithm was proposed [4]. Furthermore, the theoretical performance was analyzed in terms of statistical prediction problem [1]. The following is the brief description of the automatic detection algorithm.

1) Setting up known values

a) Choose a set of characters S_k.
b) Set a threshold as a real value $\alpha \in [0, 1]$.

2) Calculating the content rate of suspicious characters $x_{k,l}$ which is defined as,

$$x_{k,l} = \frac{\#S_k}{|\,l\,|}\,, \tag{1}$$

where $\#S_k$ denotes the size of S_k.

3) Automatic detection
Determine each input's label (normal or attack input) by the following function $d\,(x_{k,l}, \alpha)$:

$$d\,(x_{k,l}, \alpha) = \begin{cases} 0 & if \;\; x_{k,l} \leq \alpha; \\ 1 & if \;\; x_{k,l} > \alpha. \end{cases} \tag{2}$$

Eq. (2) means that if detection result is normal, then its value is zero, otherwise the result is attack and its value is one.

Example 2.1 (Automatic Detection of Attack Input): Let $l =$ "DROP sampletable;--" be an attack input where its length $|\,l\,| = 19$. Suppose that a character set S_{13} contains space, semicolon, and left round brackets. Furthermore, the threshold value α is set to 0.10.

Since the input l contains one space and one semicolon among characters in S_{13}, a numerator in Eq. (1) becomes $\#S_{13} = 2$. Therefore, according to Eq. (1),

$$\begin{aligned} x_{13,l} &= \frac{2}{19} \\ &= 0.1052\cdots \end{aligned}$$

With the above $x_{13,\,l}$, the detection result by Eq. (2) becomes the following:

$$\begin{aligned} d\,(x_{13,l}, \alpha) &= d\,(0.1052, 0.10) \\ &= 1\,. \end{aligned}$$

Thus l is detected as an attack input.

■

2.3 Performance Evaluation with Artificial Data

For evaluation of the above algorithm, the artificial data was composed [4]. Those data cover the typical types of SQL injection attack input as well as normal input among common web forms. The number of types of attack inputs was 624, on the other hand, that of normal inputs was 234. Those data were converted to the fields of single and multibytes characters, wiki, emoticon etc. These types were assumed to be input as IDs, passwords, names, and addresses etc. [4].

If the real operating situation is considered, the label of each input (either normal or attack) is unknown. Therefore, the mixture data of both labels were used for simulations in evaluations [4]. Let $0 \leq P_N \leq 1$ be the correct detection rate for normal input and let $0 \leq P_A \leq 1$ be the correct detection rate for attack input. Furthermore, if the ratio of the number of the normal input against attack input is $0 \leq \beta \leq 1$, the total detecting rate $0 \leq \mu \leq 1$ can be calculated by the following:

$$\mu\,(S_k, \alpha, \beta) = \beta P_N + (1 - \beta)\, P_A\,. \tag{3}$$

For evaluation, the value of α was empirically chosen as a constant. On the other hand, various values of β were taken with its interval $0 \leq \beta \leq 1$. For the remained S_k, the following objective function was assumed to chose the optimal character set of S^*.

$$S^* = \arg\max_{S_k} \left[\mu\,(S_k, \alpha, \beta)\right]\,. \tag{4}$$

With the above Eq. (4), the sensitive analysis was considered for discussions [4].

3. The Proposed Threshold Learning Algorithm

3.1 Formulation and Criterion

In general, the threshold value is unknown and should be learned from real observed data. If S^* has been already determined and the candidates of $\alpha_j,\; j = 1, 2, \cdots, N$ have been obtained, then the following can be defined as similar form of Eq. (4).

$$\alpha^* \;\; = \;\; \arg\max_{\alpha_j} \left[\mu\,(S_k, \alpha_j, \beta)\right]\,. \tag{5}$$

Since the label of input l is unknown at the real operation, the value of β, which is the weight of normal input against attack input is also unknown. Therefore, Eq. (4) and (5) can be achieved with several criteria. One of them is that assuming the probability distribution of $p(\beta)$ to take the expectation of $\mu\,(S_k, \alpha_j, \beta)$ with respect to β. For numerical approximation, suppose $\beta_m\,, m = 1, 2, \cdots, M$ is sampled on the interval $0 \leq \beta \leq 1$. Then, such criterion can be formulated as the following:

$$\begin{aligned} &\{\alpha^{**}, S^{**}\} \\ &\quad = \arg\max_{\alpha_j} \max_{S_k} \left[\sum_{m=1}^{M} p\,(\beta_m)\;\mu\,(S_k, \alpha_j, \beta_m)\right]\,. \end{aligned} \tag{6}$$

Note that α^{**}, S^{**} maximize the expected total detecting rate $\mu\,(S_k, \alpha_j)$ in Eq. (6).

For the other criteria, both the expected total detecting rate and the absolute value of slope of regression line can be considered which is more

restrictive. This criterion also takes into account the stability of $\mu(S_k, \alpha_j)$ with respect to β.

$$\begin{aligned}
&\{\alpha^{***}, S^{***}\} \\
&= \arg\max_{\alpha_j}\max_{S_k}\left[\sum_{m=1}^{M} p(\beta_m)\ \mu(S_k, \alpha_j, \beta_m)\right. \\
&\quad - \left|\frac{1}{M\sum_{m=1}^{M}(\beta_m)^2 - \left(\sum_{m=1}^{M}\beta_m\right)^2}\right| \\
&\quad \times \left|M\sum_{m=1}^{M}\beta_m\,\mu(S_k, \alpha_j, \beta_m)\right. \\
&\quad \left.\left. - \left(\sum_{m=1}^{M}\beta_m\right)\left(\sum_{m=1}^{M}\mu(S_k, \alpha_j, \beta_m)\right)\right|\right].
\end{aligned} \tag{7}$$

Note that α^{***}, S^{***} maximize the sum of the expected total detecting rate and the slope of regression line. In Eq. (7), the second, third, and fourth terms on the right hand side express the absolute value of slope of the regression line $\mu(S_k, \alpha_j)$ where β is its domain.

3.2 The Proposed Algorithm

With training data set that contain pairs of input sequence l and its label, the following learning algorithm of threshold value α^{**} or α^{***} would be proposed.

1) With various candidate pairs of S_k and α_j, execute automatic detection algorithm with Eq. (1) and (2).
2) Calculate the total detecting rate $\mu(S_k, \alpha_j, \beta)$ with P_N, P_A, and $0 \le \beta \le 1$.
3) Taking β_m for numerical approximation to choose the optimal set of S^{**} and α^{**} by Eq. (6)(or S^{***} and α^{***} by Eq. (7)).

4. Evaluations of the Proposed Algorithm

4.1 Conditions

For evaluations, the artificial data mentioned in subsection 2.3 was used. The number of types of attack inputs is 624, those of normal inputs is 234 where the data cover the single and multibytes characters, wiki, emocticon etc. Note that the data was assumed IDs, passwords, names, and addresses etc.

For known finite characters, the five characters were chosen as Table 1. These characters are same as our previous simulations [4].

With the above five characters, the following twenty six power sets can be defined as,

Table 1: Known Characters

Name	Character
s_i	Space
s_{ii}	Semicolon (;)
s_{iii}	Single Quotation (')
s_{iv}	Right Parenthesis ())
s_v	Left Parenthesis(()

$S_1 = \{s_i, s_{ii}\}, S_2 = \{s_i, s_{iii}\}, S_3 = \{s_i, s_{iv}\},$
$S_4 = \{s_i, s_v\}, S_5 = \{s_{ii}, s_{iii}\}, S_6 = \{s_{ii}, s_{iv}\},$
$S_7 = \{s_{ii}, s_v\}, S_8 = \{s_{iii}, s_{iv}\}, S_9 = \{s_{iii}, s_v\},$
$S_{10} = \{s_{iv}, s_v\},$
$S_{11} = \{s_i, s_{ii}, s_{iii}\}, S_{12} = \{s_i, s_{ii}, s_{iv}\},$
$S_{13} = \{s_i, s_{ii}, s_v\}, S_{14} = \{s_i, s_{iii}, s_{iv}\},$
$S_{15} = \{s_i, s_{iii}, s_v\}, S_{16} = \{s_i, s_{iv}, s_v\},$
$S_{17} = \{s_{ii}, s_{iii}, s_{iv}\}, S_{18} = \{s_{ii}, s_{iii}, s_v\},$
$S_{19} = \{s_{ii}, s_{iv}, s_v\}, S_{20} = \{s_{iii}, s_{iv}, s_v\},$
$S_{21} = \{s_i, s_{ii}, s_{iii}, s_{iv}\}, S_{22} = \{s_i, s_{ii}, s_{iii}, s_v\},$
$S_{23} = \{s_i, s_{ii}, s_{iv}, s_v\}, S_{24} = \{s_i, s_{iii}, s_{iv}, s_v\},$
$S_{25} = \{s_{ii}, s_{iii}, s_{iv}, s_v\}, S_{26} = \{s_i, s_{ii}, s_{iii}, s_{iv}, s_v\}.$

4.2 Simulations

1) Simulation 1
   Choose five pairs of $\{\alpha^{**}, S^{**}\}$ in descending order according to the criteria in Eq. (6).
2) Simulation 2
   Choose five pairs of $\{\alpha^{***}, S^{***}\}$ in descending order according to the criteria in Eq. (7).

4.3 Results

Table 2 and 3 were obtained for Simulation 1 and 2, respectively.

Table 2: Result of Simulation 1

Rank	S^{**}, α^{**}	Value in Eq. (6)
1st	$S_{22}, \alpha = 0.08$	0.9170
2nd	$S_{12}, \alpha = 0.08$	0.9154
3rd	$S_{22}, \alpha = 0.09$	0.9147
4th	$S_{21}, \alpha = 0.08$	0.9142
5th	$S_{14}, \alpha = 0.02$	0.9135

Table 3: Result of Simulation 2

Rank	S^{***}, α^{***}	Value in Eq. (7)
1st	$S_{12}, \alpha = 0.09$	0.9032
2nd	$S_{12}, \alpha = 0.08$	0.8994
3rd	$S_{22}, \alpha = 0.11$	0.8930
4th	$S_{23}, \alpha = 0.10$	0.8903
5th	$S_{21}, \alpha = 0.11$	0.8879

5. Discussions

From Table 2 in Simulation 1, we can see that the pair S_{22} and $\alpha = 0.08$ maximizes the expected total detecting rate $\mu(S_k, \alpha_j)$ in Eq. (6). In our previous research [4], the pair $S_{12}, \alpha = 0.08$ was empirically chosen. According to Table 2, relatively superior pair was discovered with the criterion in Eq. (6). Figure 1 shows the plot of the top three pairs of S^{**} and α^{**} where the vertical axis is the value of μ and the horizontal axis is $0 \leq \beta \leq 1$. According to Eq. (6), the superior μ of the pair S_{22} and $\alpha = 0.08$ can be observed comparing to the empirically chosen pair S_{12} and $\alpha = 0.08$ in Figure 1. In Figure 1, the pair S_{12} and $\alpha = 0.08$ gives the most flattest line among three lines, however, the other two pairs give the relatively superior values according to Eq. (6).

From Table 3 in Simulation 2, S_{12} and $\alpha = 0.09$ are obtained as the optimal pair according to Eq. (7). The second is the pair of S_{12} and $\alpha = 0.08$. Figure 2 shows the same sort of plot as previous Figure 1. Since Eq. (7) emphasizes the absolute value of the slope, more flatter line can be chosen. In this criterion, S_{12}, which contains three characters, was superior to the others, whereas S_{22} was superior to them in the criteria Eq. (6).

Figure 3 shows the effect of various values of α in S_{12}. From Figure 3, the more the value of α increases, the more larger the value of slope of the line becomes. Since Eq. (7) gives the penalty of the larger value of the slope, $\alpha = 0.09$ is more likely to be chosen.

Figure 4 shows the effect of various values of α in S_{21}. Figure 4 also shows that the more the value of α increases, the more the value of slope of the line becomes larger. But the increasing degree of the slope in S_{21} is relatively larger than that of S_{12}. This result can be interpreted as the effect of the character of Right Parenthesis which is the only contained character in S_{21}.

Figure 5 shows the effect of various sets among S_{12}, S_{21}, and S_{24} where those thresholds are the constant $\alpha = 0.08$. From Figure 5, the detecting performances of S_{12} and S_{21} are similar, however, that of S_{24} is the relatively poor. It can be observed that S_{24} is the only set which does not contain Semicolon.

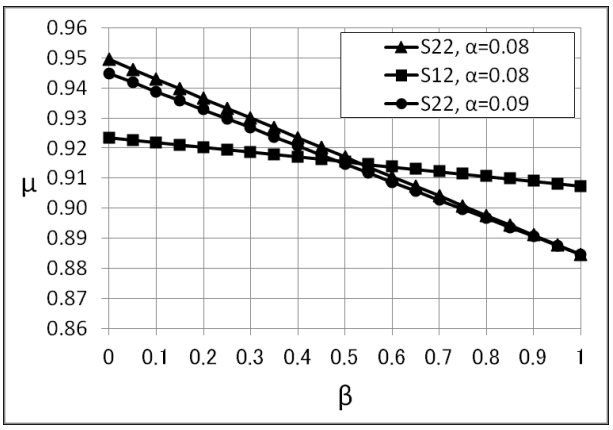


Figure 1: Learning Results based on Eq. (6)

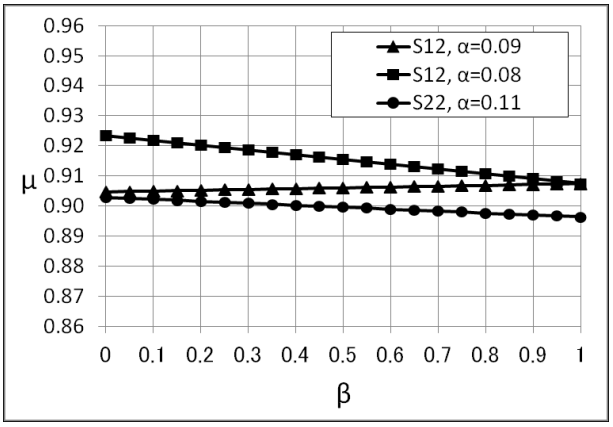


Figure 2: Learning Results based on Eq. (7)

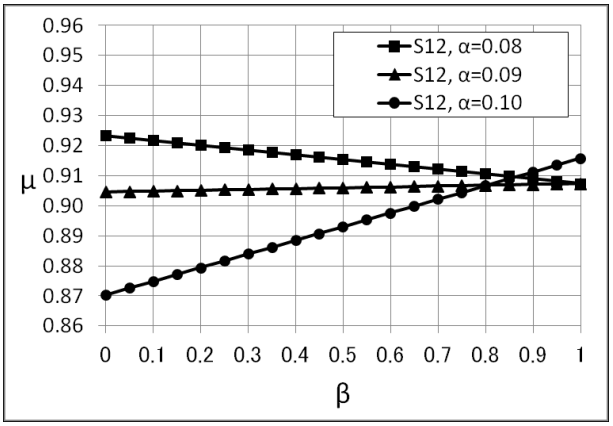


Figure 3: Detecting Performance of S_{12} with various thresholds

6. Conclusions

This paper proposed the learning algorithm to choose the real-valued threshold for automatic detection of SQL injection attack. Furthermore, some

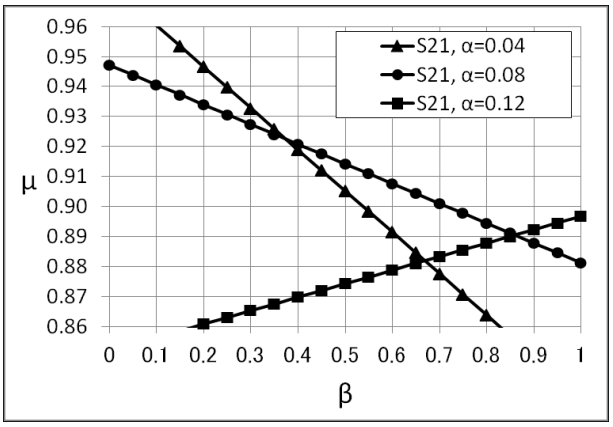


Figure 4: Detecting Performance of S_{21} with various thresholds

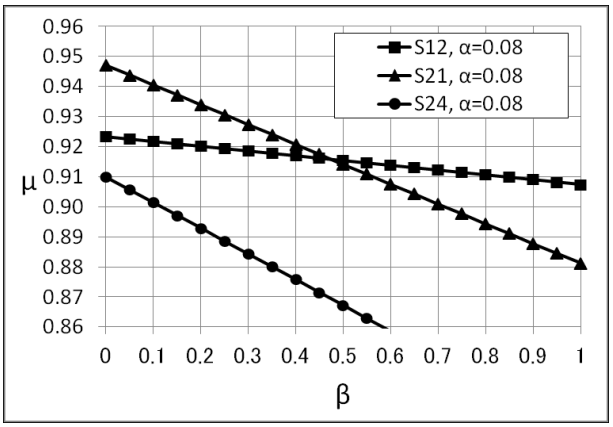


Figure 5: Detecting Performance of S_{12}, S_{21}, and S_{24} with a constant thresholds ($\alpha = 0.08$)

learning criteria were considered and their performances were also examined with artificial data. As a result, the certain effectiveness was observed with the proposed algorithm and thus seeking the unknown threshold value can be possible with training sets of data.

For future research, predictive performance should be examined with unknown data sets. Furthermore, the detecting performance with the real SQL injection data should also be considered.

Acknowledgments

This research is partially supported by the Grant-in-Aid for Scientific Research (C) No.23501178 of the Japan Society for the Promotion of Science (JSPS).

This research is partially supported by the Grant-in-Aid for Scientific Research (B) No.23740094 of the Japan Society for the Promotion of Science (JSPS).

This research is partially supported by the 44uh Kurata Grants of the Kurata Memorial Hitachi Science and Technology Foundation.

References

[1] Takeshi Matsuda, Daiki Koizumi, Michio Sonoda, and Shigeichi Hirasawa, "On Predictive Errors of SQL Injection Attack Detection by the Feature of the Single Character," Proceeding of 2011 IEEE International Conference on Systems, Man, and Cybernetics (SMC2011), pp. 1722–1727, Oct. 2011.

[2] NT Web Technology Vulnerabilities [Online], http://www.phrack.com/issues.html?issue=54

[3] The Open Web Application Security Project (OWASP), SQL Injection [Online], https://www.owasp.org/index.php/SQL_Injection

[4] Michio Sonoda, Takeshi Matsuda, Daiki Koizumi, and Shigeichi Hirasawa, "On Automatic Detection of SQL Injection Attacks by the Feature Extraction of the Single Character," Proceedings of the 4th International Conference on Security of Information and Networks (SIN2011), pp. 81–86, Nov. 2011.

Composable Network Services Framework: GÉANT Multi-domain Bus (GEMBus)

M. Grammatikou, [5] **C. Marinos,** [5]
P. Martinez-Julia[3], **J. Jofre1**[1], **S. Gheorghiu1**[1], **D. Lopez**[2], **Y. Demchenko**[7], **K. Dombek**[4], **R. Hedberg**[6], **A. Skarmeta**[3], **E. Toroglosa**[3] **and V. Pouli**[5]
1 i2CAT Foundation, Network Technologies Cluster, Gran Capità 2-4 (Nexus Building), 08034 Barcelona, Spain
2 Telefonica I+D, 28050, Madrid, Spain,
3 University of Murcia, Dept. Information and Communication EngineeringUniversity of Murcia, 30100, Murcia, Spain
4 Poznan Supercomputing and Networking Center, Noskowskiego 12/14, 60-704 Poznan, Poland
5 Network Management and Optimal Design (NETMODE) Lab, National Technical
University of Athens, Iroon Polytechneiou 9, 15780 Zografou, Greece
6 Umea University, ICT Services and System Development (ITS), 90187 Umea, Sweden,
7 University of Amsterdam, Science Park 904, 108XH Amsterdam, Netherlands

Abstract - *Service-Oriented Architecture (SOA) has become a common technology for provisioning infrastructure services on-demand. Service-Oriented Architecture (SOA) allows managing, maintaining and accessing heterogeneous and geographically sparse resources in a unified way. This paper introduces GEMBus (GEANT Multi-domain Bus), a service-oriented middleware platform that allows flexible services composition, and their on-demand provisioning and deployment to create new specialized task-oriented services and applications. GEMBus is built upon state-of-the-art Enterprise Service Bus (ESB) technologies and extend them with new functionalities that allow dynamic component services deployment, composition and management.*

The GEMBus architecture incorporates different ESB instances at different management domains, orchestrated by the GEMBus core, constituted by the elements which provide the functionality required to maintain the federation infrastructure: service registry, message infrastructure, security service, accounting service, and composition services. The current paper also discusses the general case for integration of Service-Oriented Architecture (SOA) principles and technologies with the provision and deployment mechanisms to support on-demand infrastructure services provisioning.

This architecture has been validated with a series of use-cases in the GÉANT environment and is about to be applied to similar infrastructures in a wide range of application fields within the academic and research community.

Keywords: GEMBus, SOA, federation, middleware, ESB, USDL, semantics

1 Introduction

This paper presents the framework and general architecture of GEMBus (GÉANT Multi-domain Bus), the federated multi-domain service-oriented infrastructure being developed in GN3 [1] project. GEMBus is based on a Composable Service Architecture (CSA) [2], a general framework for composite services, and on the industry-adopted Enterprise Service Bus (ESB) [3], extended to support dynamically reconfigurable virtualized services. The architecture addresses multi-domain issues and distributed heterogeneous services composition and orchestration. The possible users for GEMBus could be the network and service operators who wish to install the GEMBus platform and its core services in their administrative domain, and the users who wish to develop new services and integrate them with the GEMBus platform, enhancing their usability. Both types of users can benefit from the Composable Service Architecture.

Service-Oriented Architecture (SOA) [4] allows managing, maintaining and accessing heterogeneous and geographically sparse resources in a unified way by providing standardized interfaces and common working environments to their users. The heterogeneous nature of these resources spans not only across different providers or administrative domains, but also across different application domains, aiming, for example, at the integration of bandwidth reservation mechanisms with storage allocation procedures into what users should perceive as a single service.

The bus paradigm provides the additional advantage of freeing service developers from dealing with common aspects such as authentication, authorization, accounting, service discovery and message management. This enables them to concentrate on the direct implementation of business processes. Most current ESB frameworks are oriented to

single-enterprise deployment that relies on a central top authority. GEMBus aims to bring the advantages of these ESB frameworks into an open collaborative environment, taking a step further into federated infrastructures and supporting the definition of a multi-domain ESB infrastructure, a "bus of buses".

This paper is structured in the following way. In section 2, we describe GEMBus architecture, placing particular emphasis on GEMBus core components. In section 3, we state the successfully integrated services within the GEMBus infrastructure and in the last section finally, we present our conclusions.

2 GEMBus architecture

In Figure 1 below, is illustrated the general GEMBus architecture, where different ESB instances at different management domains are federated. Services deployed in one of the instances can seamlessly access those services made available by the others.

The GEMBus Core is comprised of those elements that provide the main functionality required to maintain the federated GEMBus infrastructure. The GEMBus Core includes specific functional components and services: GEMBus Registry, GEMBus Messaging Infrastructure (GMI), GEMBus Security Service, Composition Service and GEMBus Accounting Service.

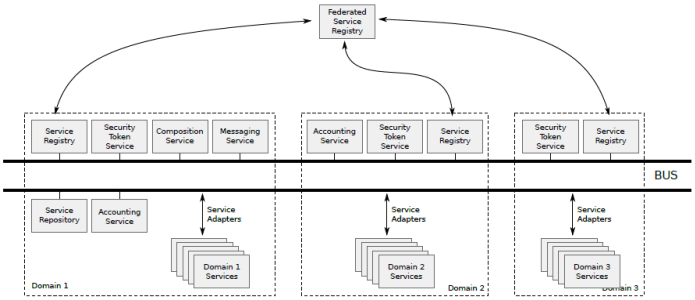


Figure 1: General GEMBus architecture, showing core elements

2.1 GEMBus Registry

In a highly dynamic and multi-domain infrastructure a mechanism to keep track of dynamic deployments is absolutely vital. The GEMBus registry is expected to provide essential functionality to serve as a global information service to keep information about both component infrastructure services and the composed services infrastructure, including also their lifecycle metadata.

Federated Registry should be able to handle all the service description formats that are available because of the variety of services. This includes, but is not limited to, Web Services Description Language (WSDL) [5], Web Application Description Language (WADL) [6] and OSGi [7], [8] bundles.

More specifically, the GEMBus Registry (Federated Service Registry) talks to the local registries to retrieve the list of services from the local domains and to announce them globally (for all participating domains). This registry has the capability to request additional information about the services. The GEMBus registry is responsible for the mapping among the initial description of requests and the requested services.

2.2 GEMBus Messaging Infrastructure (GMI)

Existing ESB frameworks and implementations incorporate a centralized model for message handling, where a domain central message processor (that also provides inter-domain message routing when required) processes all intra-domain messages. Message processors in this case act as a collapsed messaging backbone. This means that to send or receive messages, all services need to connect to one of the adapters supporting specific service related/defined message-level protocols. These adaptors are typically connected to an assigned port.

The GMI extends this functionality in the multi-domain environment considered by GEMBus. The Registry will store and provide the GMI components with service information on location, configuration and properties. These components will make use of the Registry either by using the local, internal ESB registries for local component services, or by directly querying the common GEMBus Registry service. These components will make direct use as well of the GEMBus Accounting Service facility to track message exchange in a way that is transparent to service developers and operators.

2.3 GEMBus Security Services

The GEMBus uses the Security Token Services (STS) defined in WS-Security [9] and WS-Trust [10] as a security mechanism to convey security information between services that can also be easily extended to the federated security required by the GEMBus composable services. The STS makes it possible to issue and validate security tokens, also support services (identity) federation and federated identity delegation.

In the GEMBus STS, different elements support token issues and validation, as shown in Figure 2. The Ticket Translation Service (TTS) is responsible for generating valid tokens in the system according to the received credentials, renewing and converting security tokens. Token validation is performed by the Authorization Service (AS), which can also retrieve additional attributes or policy rules from other sources to perform the validation.

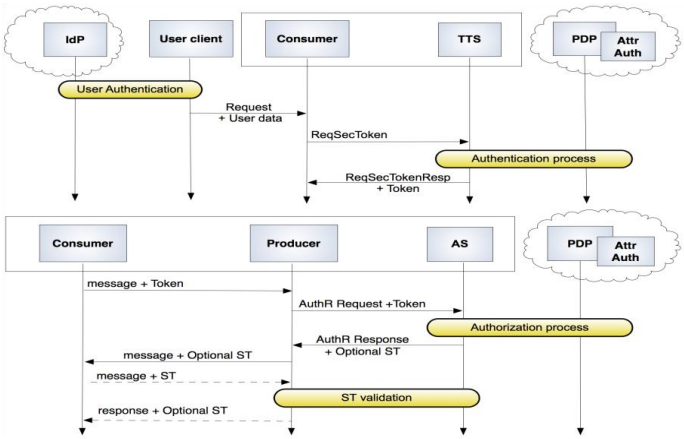


Figure 2: Authentication and authorization processes in GEMBus security services

2.4 Composition Service

In the GEMBus aim follow existing procedures and standard mechanisms, extending them to support multi-domain operation, Business Process Execution Language (BPEL) and available ESB-based execution engines are being used for service orchestration. They are connected with a specific description and control tool (first prototype implemented as an Eclipse plug-in) based on Business Process Modelling Notation (BPMN).

In addition to the orchestration service, a workflow management system will be provided by integrating the Taverna [11] environment, a cross-platform set of tools enabling users to compose not only web services, but also local Java services (Beanshell scripts), local Java APIs, R scripts and import data from Excel or in CVS format.

2.5 GEMBus Accounting Service

The GEMBus multi-domain nature requires specific mechanisms for producing and processing meaningful accounting information. The Accounting Service Architecture for GEMBus consists of a Common Accounting Repository, where all collected data from each ESB instance is stored and an Accounting Module (AM) instance is deployed at every participating ESB.

Services integrated in GEMBus use the message-oriented middleware infrastructure offered by the GMI to communicate. The function of AM is to catch and record every message exchanged (interaction between services) through the GMI, as those messages are precisely the source of information to evaluate GEMBus services behavior and performance. To capture those messages interactions exchanged through the GMI, message interceptors need to be implemented (in most of the cases, SOAP messages).

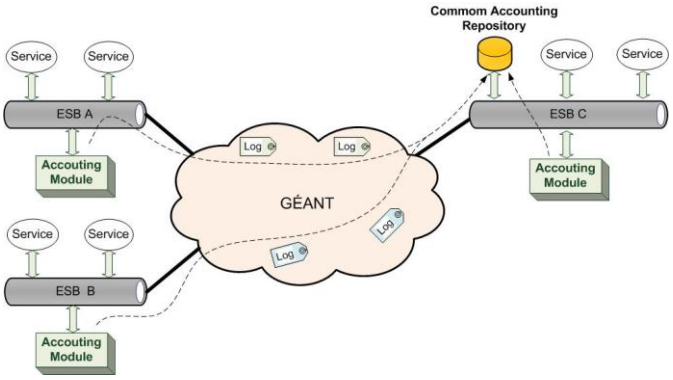


Figure: 3. Accounting Service Architecture

The accounting information will be saved using RDF ontology. RDF is an emerging language and there are several reasons for selecting RDF ontology as a data model, such as is based on open source, languages and standards. Also it can be extended and grown incrementally without impacting the existing data store. Finally, its conceptual closeness to the relational data model is an advantage. It is possible to represent RDF in a relational database and vice versa, the use of set-based semantics and queries and, via its SPARQL query language, easy mechanisms to drive faceted search and other browsing and viewing tools.

3 Using GEMBus for service integration

This section presents a use-case of service integration used during the definition of the GEMBus architecture to validate it and provide a demonstrator of its functionalities.

Use-case is related to the integration of existing GÉANT services: AutoBAHN [12] (for bandwidth on-demand provisioning) and PerfSONAR [13] (for distributed and autonomous network monitoring). Both services have followed integration patterns associated to the creation of specific adaptors to existing service interfaces, taking into account the requirements imposed by service architectures and their interactions with their supporting infrastructures.

The third case is intended to show GEMBus composition mechanisms and how they can be applied to realize an Autonomous Computing scenario. Providing Autonomic Computing (AC) capabilities within the GEMBus framework can benefit the behavior of the services and applications connected to the bus. Applications deployed on GEMBus can consume the AC services provided by the framework to gain self-management capabilities.

3.1 AutoBAHN/perfSONAR Integration as a Service

AutoBAHN service will enable the dynamic integration of bandwidth allocation mechanisms at different domains with other network services (such as monitoring services, other link

management services, and network topology and status information) and application services requiring such capabilities.

Two adaptors have been developed, providing a single entry point to the AutoBAHN infrastructure:

- The Reservation Client Adaptor, handling the user request for path reservation/cancellation/modification.

- The Reservation Service Adaptor, responsible for deploying the reservation service inside the GEMBus platform and publishing it as a service available for others to see and call it.

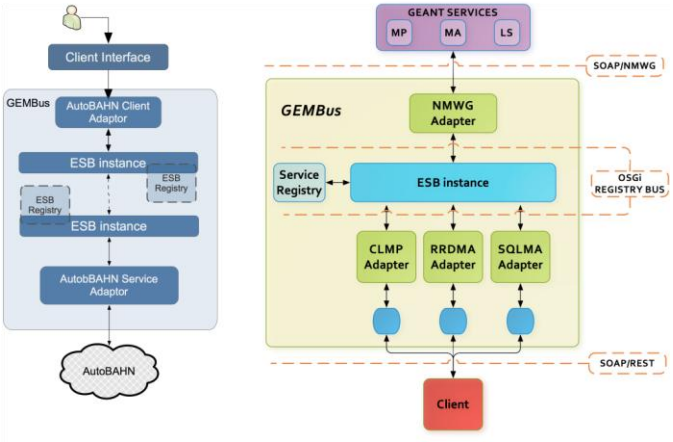


Figure 4: AutoBAHN and PerfSONAR integration within GEMBus

PerfSONAR service is based on a per-service integration pattern that takes advantage of the perfSONAR module software architecture. The per-service integration pattern assumes that each service is integrated as implementation of a separate adaptor. This solution provides several advantages such as easy management of services. Since the ESB framework manages the lifecycle of the adaptors the process related to deployment as well as stopping and restarting each of the services will be easy to perform. Having each of the services in separate adaptors also provides easy access restriction, keeping statistics of the service usage or logging all service events. This pattern offers an easy way for clients to access the services too. The usage of ESB binding elements makes it possible to put out many types of ready-to-use endpoints (such as SOAP or REST protocols) for clients, as well as easy reconfiguring.

The current version of the integration architecture prototype provides four adapters. The Network Measurement Working Group (NMWG) Adaptor is an interceptor that sends requests and receives responses to/from MA (Measurement Archive) and MP (Measurement Point) in the NMWG schema. This component is used by other GEMBus perfSONAR adapters to exchange messages between ESB and the current running services. The three other adaptors are the CLMP (Command Line Measurement Point), which is used for executing command line tools such Ping, traceroute, One-Way Active Measurement Protocol (OWAMP) and Bandwidth Test Controller (BWCTL), RRDMA (Round Robin Database Measurement Archive), which provides the capability to read measurement data stored in RRD files and SQLMA (SQL Measurement Archive).

4 Conclusions

GEMBus attempts to bring the advantages of SOA into the highly heterogeneous, open multi-domain environment that currently characterizes scientific e-infrastructures. GEMBus commitment to openness and the support of high heterogeneity allows making it useful also for environments in which a full SOA deployment is challenging by proposing and enabling some core services that can create a basis for further migration to more consistent service oriented environment.

The GEMBus architecture has been validated with a series of use-cases in the GÉANT environment and is about to be applied to similar infrastructures in a wide range of application fields within the academic and research community. The results achieved so far are encouraging and the integration of composable services in an integrated and federated infrastructure can provide a high quality of services to the academic and research community.

5 References

[1] GÉANT3 Project, www.geant.net.

[2] D. Lopez and I. Thomson 2. CSA, GEANT3 Project Deliverable DJ3.3.1: Composable Network Services use-cases (January 2010)

[3] Chappell, D., "Enterprise service bus", O'Reilly, June 2004. 247 pp.

[4] OASIS Reference Architecture Foundation for Service Oriented Architecture 1.0, Committee Draft 2, Oct. 14, 2009. http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra-cd-02.pdf

[5] WSDL, http://www.w3.org/TR/wsd/

[6] WADL, http://www.w3.org/Submission/wadl/

[7] OSGI, http://www.osgi.org/Main/HomePage

[8] RDF, http://www.w3.org/TR/rdf-schema/

[9] WS-Security, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

[10] WS-Trust, http://docs.oasis-open.org/ws-sx/ws-trust/v1.3/ws-trust.html

[11] Taverna, http://www.taverna.org.uk/

[12] AutoBAHN, http://www.geant.net/Events/ICT2010/Pages/AutoBAHNAnOverview.aspx

[13] perfSONAR,http://www.geant.net/Services/NetworkPerformanceServices/Pages/perfSONARMDM.aspx

6 Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7 2007 2013) under Grant Agreement No. 238875 (GÉANT).

Fast Algorithms for Simultaneous Optimization of Performance, Energy and Temperature in DAG Scheduling on Multi-Core Processors

Hafiz Fahad Sheikh[1] **and Ishfaq Ahmad**[1]
[1]Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX, USA

Abstract - *We have proposed two algorithms for simultaneously optimizing performance, energy, and temperature while scheduling a set of tasks on a multi-core system (PET-Scheduling). The proposed algorithms differ in the way they use task allocation and voltage selection decisions to obtain multiple schedules (trade-off solutions) with a wide range of values along each objective. PET-PPF combines a power and performance-aware allocation scheme with probabilistic voltage selection to obtain these trade-off solutions. PET-DCP, on the other hand, first adjusts the task execution times using probabilistic voltage selection before leveraging a performance-optimal scheduler to generate the final schedule. Our results on several application task graphs demonstrate that both of the proposed algorithms can obtain the trade-off curves comprising of multiple solutions to the PET-Scheduling problem. PET-DCP, however, is able to achieve identical energy and thermal improvements as that of PET-PPF but at the same time degrades the performance by as much as 1.5 times less than that of the PET-PPF.*

Keywords: dynamic thermal management, frequency allocation, multi-core systems, task scheduling, DVFS.

1 Introduction

The rapidly scaling multi-core architecture has already spanned to 100 cores on a single chip [20]. However, this quick performance gain has resulted into the complex problems of energy and thermal management. Amplified chip temperatures not only require extra efforts for cooling in the form of expensive thermal packaging or larger fan size but can also lead to problems which can affect lifespan, reliability and performance of these modern systems. For example, a higher temperature can degrade the lifespan of a system to half of its value with a nearly 10^{o}C increase in the operating temperature [2]. It has also been found that in addition to the higher temperatures, large magnitude of thermal gradients can also adversely affect the performance of the interconnects and thus can limit the performance of a system [3]. Therefore, scheduling schemes that can help to control or maintain the temperature below a given threshold are an essential requirement for the extensive use of these systems. A lot of research in the last few years has focused on temperature management, temperature-aware scheduling and performance issues related to these schemes [5], [9], [10]. Most of the research contributions target to satisfy an energy budget or a thermal constraint while minimizing the consequential performance degradation [1], [11]. While these schemes can serve to meet the imposed system-based constraints, they are unable to explore the best possible trade-off between performance and energy or performance and temperature. In addition, some important issues cannot be addressed by constraint-satisfying approaches. For example, for a certain margin of trade-off in performance, what are various improvements possible in energy and thermal profile? For different possible energy budgets and thermal constraints, what is the maximum value of performance that can be achieved? Given a set of schedules with varying values of performance, energy, and temperature, how to select the best trade-off solution? These questions demand a holistic approach for integrating performance (*P*), energy (*E*), and temperature (*T*) (*PET quantities*) into the scheduling process. For this, we address the problem of simultaneously optimizing performance, energy, and temperature while scheduling tasks on a multi-core system (*PET-Scheduling*). Such joint optimization of performance, energy, and temperature is not only complex and challenging but is also a rather unexplored problem.

PET-Scheduling problem is an aggregate of task allocation, task scheduling, and voltage selection problems with the goal of minimizing the performance, energy and temperature. We have developed novel algorithms namely PET-PPF and PET-DCP that can judiciously trade-off performance with energy and temperature. PET-PPF generates a set of probability distributions for selecting a voltage level for each task. Each distribution corresponds to a different value of expected voltage level and thus enables PET-PPF to obtain several trade-off solutions. PET-DCP, uses the same voltage selection scheme as PET-PPF, however, a significant difference is that PET-DCP performs the voltage selections before the task allocation phase. In contrast to PET-PPF, PET-DCP first updates the execution times of the tasks based on the corresponding voltage level selected for each task and then uses a performance-optimal scheduler (DCP [19]) to generate the complete schedule. The key strength of our proposed methods is that they do not aim to provide a single solution to the problem. Rather several trade-off solutions are determined for the *PET-Scheduling* problem. This approach to the *PET-Scheduling* problem is correct as in the presence of multiple conflicting objectives one solution can dominate the other along

different objectives. And for such cases, where all quantities are equally important, one quantity cannot be directly preferred over the other.

The rest of the paper is organized as follows: Section II covers the related work on energy and thermal-aware scheduling. Section III presents the details of the problem under consideration. Section IV explains the proposed algorithms for the solution of the problem and Section V highlights the evaluation setup. Section VI explains the results of the simulation while Section VII concludes the paper.

2 Related Work

Most of the research efforts in the energy and thermal-aware scheduling, target to satisfy a given thermal or energy constraint at the cost of some loss in performance. Primarily, dynamic voltage and frequency scaling (DVFS) is used to adjust the voltage levels of the cores to reduce the power consumption and thus the energy and temperature. The methods in [6], [9], [10], [15] aim to meet the thermal constraints for different kinds of workloads and systems under consideration. A solution for maximizing performance under the imposed power and thermal constraints, by solving the frequency assignment problem for multi-core systems is presented in [9]. Another frequency planning method leveraging combinatorial optimization framework to maximize performance of multi-core systems with thermal limits is developed in [10]. An Event based scheduling method that can improve the peak temperature along with the total number of DTMs without excessive computation overhead is presented in [15]. The scheme proposed in [6] uses a non-DVFS approach to calculate the optimal core states for the given thermal constraints. For tasks represented as DAGs (Directed Acyclic Graphs), several iterative techniques in terms of performance degradation and computational complexity are compared in [11]. These techniques aim to find the most suitable task for the voltage adjustment to satisfy the given thermal constraint. There are also several methods that handle DTM with an alternate perspective, instead of considering temperature as a form of constraint, they try to improve thermal profile of the system within the allowed performance margins [5], [8], [14]. Similarly, there are numerous research efforts which aim to meet the given energy budget while sacrificing as little performance as possible. A power shifting method in [25] that controls the power of the different components of a server system is shown to improve the power budget based on workload conditions. A method to minimize schedule length under an energy constraint by determining the optimal power supplies on each processor is outlined in [18].

However, there are far lesser number of contributions in the pursuit of joint optimization of performance and energy or performance and temperature. The scheme presented in [4] minimizes the energy consumption and performance penalty leveraging Complier-Driven techniques. A hybrid hardware-software approach can improve performance loss consequential to the application of DVFS by leveraging reconfigurable fetch, issue, and retirement units etc., without exceeding the thermal limit [12]. The impact on the thermal profiles of cores due to the power activation at different locations of the chip represented as look-up tables have been used in [16] to allocate tasks to different cores. The proposed approach targets to improve the peak temperature as well as guarantee the thermal limit while at the same time decreases the rejection ratios under thermally constrained CMPs. A few research efforts report improvements in performance and energy under an efficient DTM policy [13]; however, these improvements are usually the by-products of the thermal management.

In contrast to the above mentioned research contributions we have targeted the simultaneous optimization of performance, energy, and temperature for allocating tasks on a multi-core system. We have compared our solutions to the schedules which only target to achieve maximum performance and do not take energy and temperature into consideration. This comparison can help to quantify the actual performance loss exhibited by various solutions in the pursuit of energy and temperature minimization.

3 PET-Scheduling Problem

Given a task graph with N tasks, the total number of cores (M) and the set of available voltage levels ($\boldsymbol{L}$), we aim to minimize makespan, energy consumption and temperature simultaneously while solving the task allocation, task scheduling, and voltage selection problem for the given task set. Thus the required objectives are:

$$Minimize \quad \max_{1 \le i \le N} ft_i \tag{1}$$

$$Minimize \quad \sum_{i=1}^{N} P_i . et_i \tag{2}$$

$$Minimize \quad \max_{1 \le i \le N} \max_{1 \le j \le M} T_i^j \tag{3}$$

ft_i represents the finish time of the ith task. P_i is the power dissipated during the execution of the ith task and et_i is corresponding execution time. T_i^j is the temperature of the jth core during the execution of the ith task (details of thermal model will be presented in Section V). Our goal is not to only produce one solution for solving the above mentioned min-min-min problem but to explore the whole Pareto front that exists between performance, energy, and temperature (*PET quantities*). These trade-off solutions can be used to guide the overall scheduling process to meet the required objectives. For workload, we considered tasks with precedence relationships represented as directed acyclic graphs (DAGs). Several scientific and multimedia applications can be conveniently represented as DAGs. A DAG consists of weighted nodes and edges, where the

weight of the node represents the cost associated with the computation of the task and the weight on an edge represents the communication cost between the two tasks or nodes. Critical path in a DAG is defined as the path of longest length in the graph and hence governs the latest finishing time of a scheduled DAG [7]. The nodes constituting critical path are known as critical path nodes (CPNs) and the cores on which these tasks are scheduled are called the critical cores. Nodes having successors on critical path are known as In-bound nodes (IBNs). All other nodes are called out-bound nodes (OBNs) [7].

4 Proposed Solution

We will explain the algorithms proposed for solving the *PET-Scheduling* problem by highlighting their task allocation and voltage selection phases followed by their computational complexities. While doing so, it is assumed that there are *M* cores in the system which can switch across *K* voltage levels and the DAG to be scheduled has *N* tasks/nodes.

4.1 PET-PPF

For DAG scheduling, the decision space for the *Pet-Scheduling* problem spans not only the task allocation decisions but also include task ordering and voltage selection decisions. PET-PPF solves the problem in a hierarchical manner. For task allocation, PET-PPF aims to minimize the product of total power consumption of the cores and their available time for allocating the upcoming task. The intuition is to include performance and power directly into the allocation decisions as both energy consumption and temperature are related to the power dissipation. In other words, while allocating ith task we select the core with minimum PP^j_i which is defined as:

$$PP_i^j = s_{i-1}^j \, P_{i-1}^j, \;\; \forall 1 \le j \le M \tag{4}$$

where, s_{i-1}^j represents the finish time of the jth core after allocating i-1 tasks to all the cores. P_{i-1}^j is the total power consumption of the jth core just before allocating ith task. Therefore, ith task is allocated to the core such that:

$$y_{i,j} = \begin{cases} 1 & for\ j = \arg\min_{1 \le \delta \le M} (PP_i^{\delta}) \\ 0 & otherwise \end{cases} \tag{5}$$

In (5), $y_{i,j}$ is set to 1 if ith task is allocated to the jth core. The makespan of a scheduled DAG varies significantly with how tasks are prioritized during the allocation phase. We assigned priorities to the tasks according to their classification in the DAG. CPNs are given the highest priority followed by IBNs while OBNs are kept at the lowest priority. The list of tasks generated by this classification is usually termed as CPN Dominant Sequence (CPN-DS) [7]. However, while constructing CPN-DS the precedence constraints are evaluated to ensure that the parent nodes are added to the list prior to the task itself. In voltage selection

PET-PPF

```
1: K=Total number of voltage levels
2:Initialize P_dist ← uniform distribution for voltage selection
3:// First Shifting the peak towards highest voltage level
4:pivot_ point ← select a partition point on the set of voltage levels
5:for given number of adjustment steps (τ)
6:   for all levels from start to pivot_point
7:      reduction ← P_dist(level) / reductionPerStep (η)
8:      P_dist(level) ← P_dist(level) -reduction
9:      creduction+= reduction
10:  end for
11:  for all levels from pivot_point to K
12:     P_dist(level) = P_dist(level)+ proportional factor*creduction
13:     // Higher voltage levels get large components of creduction
14:  end for
15:  Vlevels =generateVlevels( P_dist)
16:  for all tasks ϵ DAG (V, E)
17:     selected_core ← find the core with minimum TP_product
18:     Allocate task to selected_core at the earliest possible time ST
19:     udateSystemState();
20:  endfor
21:  updateSolutionSet(currentSchedule);
22:end for
23:Initialize P_dist ← 1/K
24:// Shifting the peak towards lowest voltage level
25:for given number of adjustment steps (τ)
26:  for all levels from pivot_point to K
27:     reduction ← P_dist(level) / reductionPerStep (η)
28:     P_dist(level) ← P_dist(level) -reduction
29:     creduction+= reduction
30:  end for
31:  for all levels from start to pivot_point
32:     P_dist(level) = P_dist(level)+ proportional factor*creduction
33:     // Lower Voltage levels get large components of creduction
34:  end for
35:  Vlevels =generateVlevels( P_dist)
36:  for all tasks ϵ DAG (V, E)
37:     selected_core ← find the core with minimum TP_product
38:     Allocate task to selected_core at the earliest possible time ST
39:     updateSystemState();
40:  endfor
41:  updateSolutionSet(currentSchedule);
42:endfor
```

Figure 1: PET-PPF

phase, a set of probability distributions is generated. Each probability distribution is used to select the voltage level for every task in the DAG, thus generating a potentially different schedule in the objective domain. In other words, to obtain τ trade-off solutions, we generate τ probability distributions. For generating this set of distributions, we start with a uniform distribution (allowing each voltage level to have the equal chance of getting selected for every task). We then transform the distribution in each step to first shift the peak of distribution towards the maximum voltage level and then repeat the procedure starting with uniform distribution to shift the peak of distribution towards the lowest available voltage level. In other words, we start with a uniform probability distribution in first step, as:

$$\Pr(x = L_i) = \frac{1}{K}, \;\; \forall L_i \in \mathbf{L} \tag{6}$$

In the above equation **L** represents the set of available voltage levels. Now, to adjust the peak of this distribution we define a *partition index* on the set **L**. Such that in each

PET-DCP

```
1:L=loadVoltageLevels
2:distributionData=generateProbDist()
3:for n=1 to size(distributionData)
4:   currentprob=distributionData(n);
5:   vLevels=generateVlevels(currentprob);
6:   currenttaskgraph =updateTaskGraph(filename,currentprob);
7:   newshedule=dcpSchedule(currenttaskgraph);
8:   makespan=getLatestCompletiontime(newschedule);
9:   energyconsumption=
10:  getEnergyConsumption(newschedule,vLevels)
11:  maxTemp=max(getThermalProfiles(newschedule,vLevels))
12:  updateSolutionSet(currentSchedule);
13:endfor
```

Figure 2**:** PET-DCP

transformation step the probabilities corresponding to the voltage levels with indexes up to the *partition index* are reduced by a certain factor and the collective reduction is then distributed to the probabilities of voltage levels present in the second partition. If α represents the *partition index* over the set of available voltage levels and η defines the fractional reduction in the probability values corresponding to the selected voltage levels. Then the new distribution can be given by:

$$\Pr(L_i)=\begin{cases}\Pr(x=L_i)/\eta & \forall\, i\le\alpha\\ \Pr(x=L_i)+\dfrac{L_i}{\sum_{r=\alpha}^{K}L_r}\sum_{m=1}^{\alpha}(\eta-1)\Pr(x=L_m) & \forall\ \alpha\le i\le K\end{cases}\qquad(7)$$

In the above equation the parameters α and η can be adjusted to control the computational complexity of the overall approach. We found empirically that with α set to the middle of set **L** and η=2, several unique distributions can be obtained when size of **L** is not very large ($|\mathbf{L}| \le 10$). It must be noted that (7) represents the adjustments done for gradually shifting the probability distribution to favor the maximum voltage level and may be repeated τ times, where τ can be selected based on the value of η. In addition, the increase in the probability of selection for each level is proportional to the value of its voltage. The required modification is straight forward for the case where we need to favor the lowest voltage level. Figure 1 presents the overall procedure used by the PET-PPF.

4.1.1 Computational Complexity of PET-PPF

The total number of adjustments steps can be controlled by τ for each direction, therefore, the probability distribution adjustment phase is $O(\tau K)$. In the task allocation phase, a single term as in (4) is evaluated for every core per each task. Hence, the complexity of task allocation phase is $O(M)$. So, for a DAG with N tasks, the overall complexity of PET-PPF is $O(\tau N (M+K))$.

4.2 PET-DCP

PET-DCP takes an opposite approach to that of the conventional DTM and energy improvement schemes. Most of such schemes adjust a performance-optimal schedule to satisfy the given thermal and energy requirements. However, PET-DCP, starts with the task adjustment phase using the given trade-off margin, and then leverages a performance-optimal scheduler to generate the final schedule. In other words, initially a set of probability distributions (similar to PET-PPF) for voltage selection phase is generated. Each distribution is then used to select the voltage level for every task in the DAG. Based on the selected voltage levels, the execution time of each task is updated. This updated DAG is then used as input to the DCP (Dynamic Critical Path) scheduler [21] which generates the final schedule. DCP is a performance-optimal scheduler which keeps track of the critical path after every task allocation and assigns priorities to the remaining tasks accordingly. DCP has been shown to generate schedules with near-optimal makespan [7]. As the expected voltage level varies across the probability distributions generated in the voltage selection phase. Therefore, each distribution allows a different level of performance trade-off which translates into possibly different energy and thermal improvements. It should be noted that any change in the execution time of tasks can result into a possible modification of the critical path and thus the initial schedule as generated by DCP no longer remains optimal. Therefore, starting from a performance-optimal schedule and then iteratively updating it for the desired energy and thermal requirements may potentially lose more performance in the pursuit of energy and temperature improvements. However, the use of performance-aware scheduler by PET-DCP as a second step ensures maximum performance for the modified task graph. Thus, PET-DCP can potentially achieve the performance-energy and performance-temperature trade-offs without excessive performance degradation. Figure 2 briefly outlines the PET-DCP approach.

4.2.1 Computational complexity of PET-DCP

The computational complexity of DCP to generate a schedule for a DAG of N tasks is $O(N^3)$. The total number of adjustments steps are 2τ, therefore the complexity of probability distribution adjustment phase is $O(\tau K)$. Ignoring the time-complexity of drawing N levels from the given distribution ($O(KN)$) and the time to update the DAG ($O(N)$), the overall complexity of PET-DCP will be $O(\tau K N^3)$.

5 Experimental Details

We assumed a 16-core system with cores arranged in a grid layout of 4x4. However, the proposed approach can be used for any number of cores and voltage levels. Each core was assumed to be able to switch across 5 different voltage levels in active mode, thus changing the power consumption and frequency of the system. The values of frequencies at different voltage levels along with their power consumption are outlined in Table 1. It should be noted that the frequency-power scaling relationship used in our evaluation

TABLE 1
DVFS PARAMETERS

f(MHz)	1600	2000	2200	2400	2600
P(W)	23.61	48.90	72.48	93.12	105.00

TABLE 2
SYSTEM PARAMETERS

Parameter	No. of Cores	Layout	Freq. Switching	Partition index (α)	Adj. factor/step (η)	Total adj. steps (τ)	Total no. of solutions
Value	16	Grid 4x4	Independent	3	2	10	27

is not very aggressive in terms of reducing the power with the change in frequency/voltage level. Similar scaling relationships have been observed by other research efforts based on actual multi-core systems [17]. Various other parameters related to the system under consideration and the proposed algorithms are listed in Table 2.

5.1 Thermal model

To estimate the temperature of the cores with various power dissipation levels, we can use a steady state thermal model as:

$$T_j = R_{th} P_j + T_A \quad (8)$$

In the above equation, T_j represents the temperature of the jth core due to a power dissipation of P_j watts. R_{th} is the thermal resistance and T_A represents the ambient temperature. Though the model in (8) has been frequently used in various DTM related research efforts, however, it does not take into account the power consumption of the neighboring cores while calculating the temperature of each core. In order to cater for the power dissipation of the neighboring cores we can modify (8), similar to [24] as:

$$T_j = R_{th} P_j + \sum_{\forall m \in neighbor_j} \gamma R_{th} P_m + T_A \quad (9)$$

In (9), $neighbor_j$ represents the set of cores which are adjacent to the jth core. The correlation between the power consumption of the neighboring cores and the temperature of a particular core can be controlled by γ. Though the model in (9) is still simplistic, however, it should be noted that any thermal and system model can be used in conjunction with the proposed algorithms. Since, the presented algorithms provide a mechanism to explore trade-off surfaces that exist between performance, energy, and temperature, therefore, the values of these quantities can be obtained from any complex and detailed models without having an impact on the results reported in Section VI.

5.2 Task model

We used task graphs of various applications including Fast Fourier Transform [22], Laplace Equation [23], Gauss Elimination [22], Fpppp [21] and a Robot Control application [21]. Details of these task graphs can be found in the provided references.

6 Results

Figures 3-5 compare the trade-off regions obtained by the proposed algorithms for various application task graphs. While figures 6-8 present the corresponding performance-

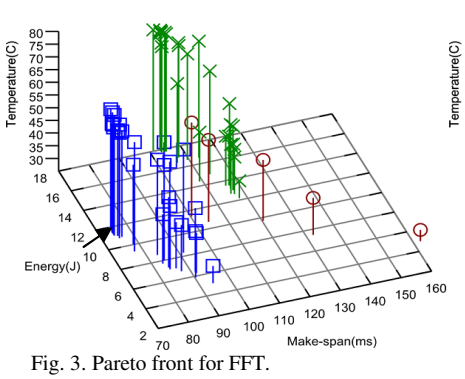


Fig. 3. Pareto front for FFT.

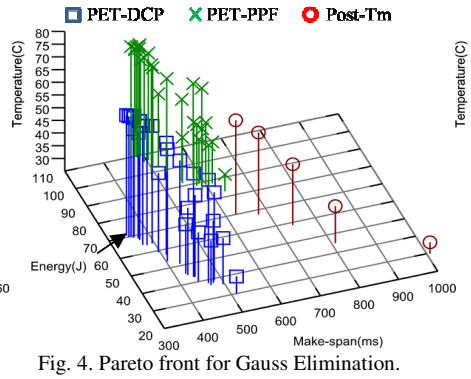


Fig. 4. Pareto front for Gauss Elimination.

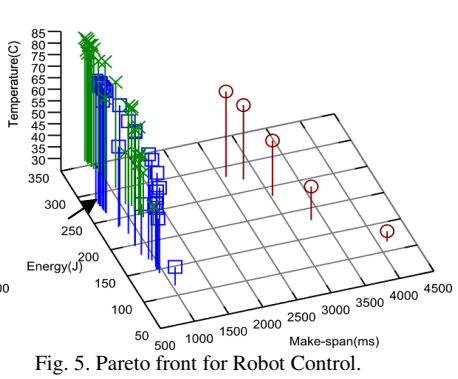


Fig. 5. Pareto front for Robot Control.

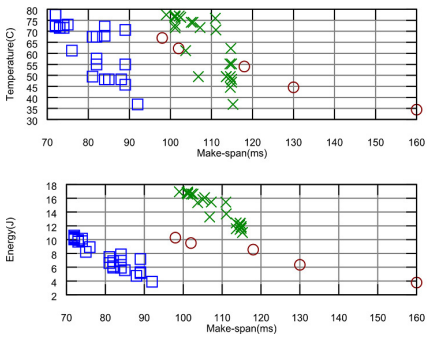


Fig. 6. Trade-off curves for FFT.

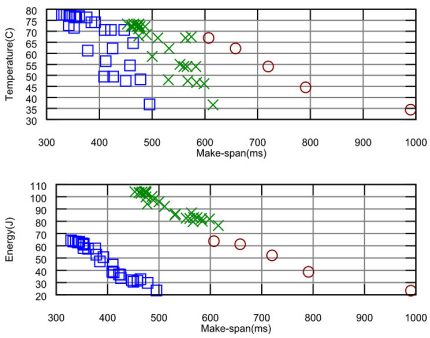


Fig. 7. Trade-off curves for Gauss Elimination.

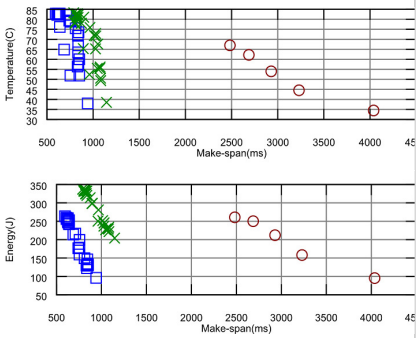


Fig. 8. Trade-off curves for Robot Control.

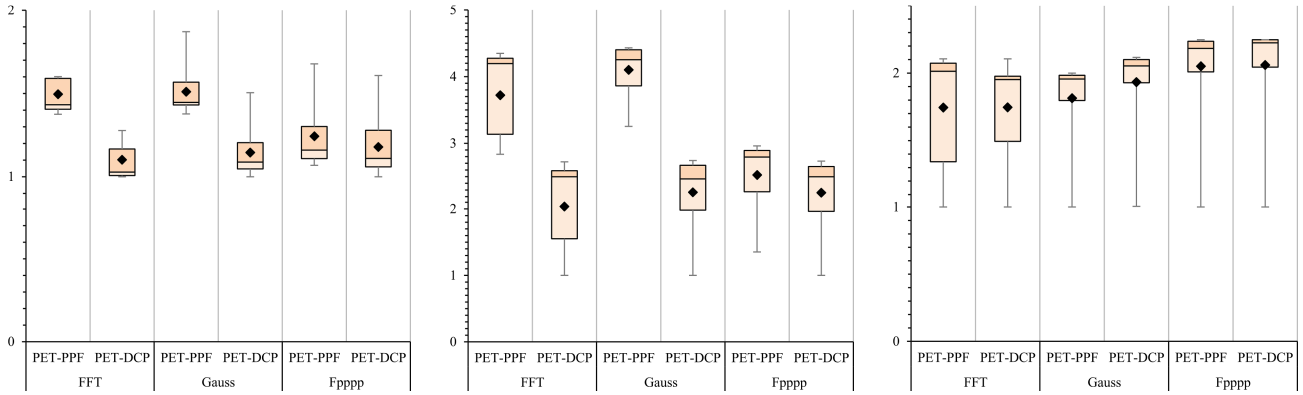


Fig. 9. Comparison of PET-PPF and PET-DCP for various task graphs along (a) Performance, (b) Energy, and (c) Peak temperature.

energy and performance-temperature trade-offs possible by leveraging the Pareto-fronts obtained from the proposed algorithms. Due to the space considerations, we have only presented the selected figures and tables. For comparison, we used an efficient temperature-aware allocation scheme called Post-Tm [16]. We used each of the available voltage level from Table 1 along with the Post-Tm allocation scheme to generate a base-line trade-off surface. A direct comparison of these Pareto surfaces (Figures 3-8) proves that both of the proposed algorithms can obtain trade-off solutions with wide range of values and with better spread along each objective as compared to the modified Post-Tm approach. However, a contrasting difference is that PET-DCP is able to generate the Pareto fronts or trade-off surfaces which are much closer to the performance optimal point (shown by arrows in Figures 3-5). Since, the spread of solutions generated by the PET-DCP is not inferior to both PET-PPF and Post-Tm along energy and temperature axis, therefore, the closeness to origin along performance axis translates into energy-performance and temperature-performance trade-offs with lesser performance degradation. Figure 9 compares the algorithms in terms of the distribution of the solution points (normalized to the minimum value obtained) along each objective. We can observe that PET-DCP was able to achieve a range of values comparable to PET-PPF but on the other hand, attains a significantly lower mean value for most of the cases.

To further analyze the quality of trade-offs, Table 3 compares the amount of performance degradation for the corresponding improvements in energy and temperature for each algorithm. While calculating the performance degradation as well as energy and temperature reductions for each trade-off solution, the values of *PET quantities* corresponding to the performance-optimal schedule as generated by DCP were used as reference. $\Delta P_m/\Delta E_m$ represents the ratio of percentage performance degradation to the percentage decrease in energy and $\Delta P_m/\Delta T_m$ is the ratio of percentage performance degradation to the percentage reduction in peak temperature (averaged over all solutions generated by each algorithm for a given task graph). Negative values for $\Delta P_m/\Delta E_m$ and $\Delta P_m/\Delta T_m$ in Table 3 represent an average decrease in the energy and temperature consequential to the corresponding performance degradation. From the values in Table 3, it can be observed that both the PET-PPF and Post-Tm degrades the performance by a larger percentage than the corresponding percentage decrease in energy and temperature. For example, for Gauss Elimination task graph the values of $\Delta P_m/\Delta T_m$ for PET-PPF and Post-Tm are -3.04 and -3.99 which means that for every 1% reduction in peak temperature, performance has to be degraded by 3.04% and 3.99% respectively. However, PET-DCP needs to degrade the performance only by 1.43% for every 1% improvement in peak temperature for the same application. Similar comparison exists for other task graphs. Figure 10 highlights this trend pictorially for all the task graphs used in our experiments. We also observe that PET-PPF, yields positive values for $\Delta P_m/\Delta E_m$ for some of the task graphs. This points out that, PET-PPF is unable to obtain large number of trade-off solutions that can improve energy consumption as

TABLE 3
TRADE-OFF RATIOS

Application	PET-PPF		Post-Tm		PET-DCP	
	$\Delta P_m/\Delta E_m$	$\Delta P_m/\Delta T_m$	$\Delta P_m/\Delta E_m$	$\Delta P_m/\Delta T_m$	$\Delta P_m/\Delta E_m$	$\Delta P_m/\Delta T_m$
Fft	1.11	-2.89	-2.96	-2.13	-0.49	-0.59
Gauss	1.31	-3.04	-5.00	-3.99	-0.77	-1.43
Laplace	1.75	-2.03	-5.29	-4.16	-0.90	-1.36
Robot	7.89	-3.84	-16.08	-11.40	-0.87	-1.85
Fpppp	-2.05	-2.56	-28.27	-17.82	-0.94	-2.07
Average	2.00	-2.87	-11.52	-7.90	-0.79	-1.46

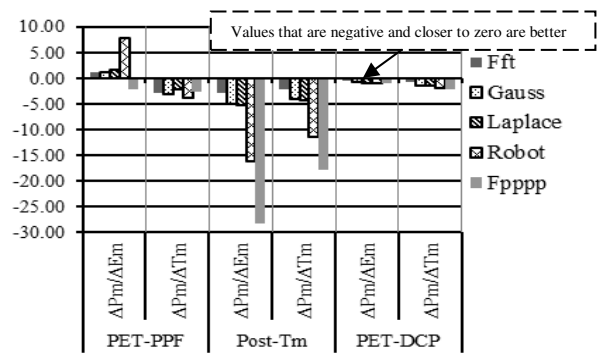


Fig. 10. Comparison of Trade-off Ratios.

compared to the base-line performance-optimal schedule.

Based on the performance of PET-DCP, we can note that in general, it may be potentially better to pre-adjust the tasks according to the system's state and the given requirements of energy and temperature before targeting the maximal performance to obtain better trade-off solutions. It may also be noted that once the trade-off solutions are available, an operating point or schedule can be selected from them according to the imposed constraints and the given preferences.

7 Conclusion

We proposed two schemes to explore the trade-off regions that may exist while trading loss in performance with the improvements in energy and peak temperature. Both algorithms were able to generate trade-off curves comprising of schedules that result into diverse range of values for makespan, energy consumption and peak temperature. Our evaluation results indicate that PET-DCP, which pre-adjusts tasks probabilistically for energy and thermal improvements before using a performance-optimal scheduler, can produce multiple schedules that are very close to the performance-optimal point. This leads the PET-DCP to achieve *trade-off ratios* better than the other algorithms (PET-PPF and Post-Tm) by a factor of 2 on average. The work presented in this paper is an inaugural effort to jointly optimize performance, energy, and temperature while scheduling tasks on a multi-core system and can be used as a framework to attain efficient trade-offs among the *PET quantities*.

8 References

[1] D. King, I. Ahmad, H.F. Sheikh, "Stretch and compress based re-scheduling techniques for minimizing the execution times of DAGs on multi-core processors under energy constraints," *Green Computing Conference, 2010 International*, pp.49-60, 15-18 Aug. 2010

[2] R. Viswananth, V. Wakharkar, A. Watwe, and V. Lebonheur, "Thermal Performance Challenges from Silicon to Systems," *Intel Technol. J., Q3,* vol. 23, p. 16, 2000.

[3] A. H. Ajami, K. Banerjee, and M. Pedram, "Modeling and Analysis of Nonuniform Substrate Temperature Effects on Global Ulsi Interconnects," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 6, pp. 849-861, Jun. 2005.

[4] U. Kremer, J. Hicks, and J.M. Rehg, "Compiler-Directed Remote Task Execution for Power Management," Workshop on Compilers and Operating Systems for Low Power (COLP), 2000.

[5] N. Fisher, J. Chen, S. Wang, and L. Thiele, "Thermal-Aware Global Real-Time Scheduling on Multicore Systems," *15th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2009.

[6] R. Rao and S. Vrudhula, "Efficient Online Computation of Core Speeds to Maximize the Throughput of Thermally Constrained Multi-core Processors," Proceedings of the International Conference on Computer-Aided Design, November 2008.

[7] Y-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.* vol 31, no. 4, pp. 406-471, 1999.

[8] A. Coşkun, T. Rosing, K. Whisnant, and K. Gross, "Static and Dynamic Temperature-Aware Scheduling for Multiprocessor SoCs," *IEEE Transactions On Very Large Scale Integration (VLSI) Systems*, Vol. 16, No. 9, September 2008.

[9] T. Chantem, R. Dick, and X. Hu, "Temperature-Aware Scheduling and Assignment for Hard Real-Time Applications on MPSoCs," *Proceedings of the Conference on Design, Automation and Test in Europe*, March 2008.

[10] A. K. Coskun, T. S. Rosing, and K. C. Gross, "Proactive Temperature Management in MPSoCs," *Proceeding of the 13th international Symposium on Low Power Electronics and Design, ISLPED '08*, August 2008.

[11] H.F. Sheikh, I. Ahmad, "Fast algorithms for thermal constrained performance optimization in DAG scheduling on multi-core processors," *Green Computing Conference and Workshops (IGCC), 2011 International* , pp.1-8, July 2011.

[12] O. Khan and S. Kundu, "Hardware/Software Co-design Architecture for Thermal Management of Chip Multiprocessors," *Conference & Exhibition on Design, Automation & Test in Europe, DATE '09*, April 2009.

[13] T. Ebi, M. Faruque, and J. Henkel, "TAPE: Thermal-aware Agent-based Power Economy Multi/many-Core Architectures," *IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers, ICCAD 2009*, pp.302-309, November 2009.

[14] D. Puschini, F. Clermidy, P. Benoit, G. Sassatelli, and L. Torres, "Temperature-Aware Distributed Run-Time Optimization on MP-SoC Using Game Theory," *Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual*, pp.375-380, April 2008.

[15] J. Cui, D.L. Maskell, "High Level Event Driven Thermal Estimation for Thermal Aware Task Allocation and Scheduling," *15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010, pp.793-798, 18-21 Jan. 2010.

[16] J. Cui and D. L. Maskell, "Dynamic Thermal-Aware Scheduling on Chip Multiprocessor for Soft Real-Time Systems," Proceedings of the 19th ACM Great Lakes Symposium on VLSI, GLSVLSI '09, May 2009.

[17] R. Cochran, C. Hankendi, A. Coskun, S. Reda, "Identifying the optimal energy-efficient operating points of parallel workloads," *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on* , pp.608-615, Nov. 2011.

[18] K. Li, "Performance Analysis of Power-Aware Task Scheduling Algorithms on Multiprocessor Computers with Dynamic Voltage and Speed," IEEE Transactions on Parallel and Distributed Systems, pp. 1484-1497, November, 2008.

[19] Y-K. Kwok, I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on* , vol.7, no.5, pp.506-521, May 1996.

[20] Tilera, "Tile-GX3100", [Online]. Available: http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx 3000 Series Brief.pdf.

[21] Standard Task Graph Set, "STG", [Online]. Available: http://www.kasahara.elec.waseda.ac.jp/schedule/

[22] I. Ahmad, Y.-K. Kwok, M.-Y. WU, and W. Shu, "CASCH: A Tool for Computer-Aided Scheduling," IEEE Concurrency, vol. 8, no. 4, pp. 21-33, October 2000.

[23] M.-Y. Wu, D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," IEEE Transactions on Parallel and Distributed Systems, vol.1, no. 3, pp. 330-343, July 1990.

[24] Z. Wang and S. Ranka, "A Simple Thermal Model for MPSoC and its Application to Slack Allocation," *Proceeding of IEEE International Parallel & Distributed Processing Symposium*, 2010.

[25] W. Felter, K. Rajamani, T. Keller, and C. Rusu, "A performance-conserving approach for reducing peak power consumption in server systems," *In Proceedings of the 19th annual international conference on Supercomputing* (ICS '05). ACM, 2005.

Parallelization Strategies for Local Search Algorithms on Graphics Processing Units

Audrey Delévacq, Pierre Delisle, and Michaël Krajecki
CReSTIC, Université de Reims Champagne-Ardenne, Reims, France

Abstract— *The purpose of this paper is to propose effective parallelization strategies for Local Search algorithms on Graphics Processing Units (GPU). We consider the distribution of the 3-opt neighborhood structure embedded in the Iterated Local Search framework. Three resulting approaches are evaluated and compared on both speedup and solution quality on a state-of-the-art Fermi GPU architecture. Solving instances of the Travelling Salesman Problem ranging from 100 to 3038 cities, we report speedups of up to 8.51 with solution quality similar to the best known sequential implementations and of up to 45.40 with a variable increase in tour length. The proposed experimental study highlights the influence of the pivoting rule, neighborhood size and parallelization granularity on the obtained level of performance.*

Keywords: TSP, ILS, Parallel Metaheuristics, 3-opt, GPU

1. Introduction

Iterated Local Search (ILS) is a metaheuristic that successively applies a given Local Search (LS) procedure to an initial solution and incorporates mechanisms to climb out of local optima. This method finds good solutions to many optimization problems in a reasonable time which may still remain too high in practice. However, it offers an interesting parallelization potential when extended to a population-based approach where different individuals improve their solution by executing the same algorithm on several computing units [1].

At present time, the best known implementations of the ILS framework are dedicated to conventional, CPU-based sequential and parallel architectures. However, as recent years have shown the potential to use the GPU as a high performance computational resource, it becomes important to deliver GPU-based optimization methods that are efficient in both solution quality and execution speed. However, recent research has shown that this goal is often difficult to achieve.

The purpose of this paper is to propose parallelization strategies for ILS to efficiently solve the Travelling Salesman Problem (TSP) in a GPU computing environment. The 3-opt exchange procedure used within the algorithm is decomposed on processing elements along with required data structures. Pivoting rules based on first-improvement and best-improvement schemes are also evaluated. Important algorithmic, technical and programming issues that may be encountered in this context are finally highlighted.

This paper is organized as follows. First, we present k-opt LS algorithms, the ILS metaheuristic and their application to the TSP. After a fairly complete review of the literature on parallel LS and ILS, the proposed GPU parallelization strategies are explained. Finally, a comparative experimental study is performed to evaluate the performance of the resulting algorithms.

2. Iterated Local Search for the Travelling Salesman Problem

The Travelling Salesman Problem (TSP) may be defined as a complete weighted directed graph $G = (V, A, d)$ where $V = \{1, 2, ..., n\}$ is a set of vertices (cities), $A = \{(i, j) \mid (i, j) \in V \times V\}$ is the set of arcs and $d : A \rightarrow \mathbb{N}$ is a function assigning a weight or distance d_{ij} to every arc (i, j). The objective is to find a minimum weight Hamilton cycle in G, which is a tour of minimal length visiting each city exactly once.

Local Search (LS) generally aims to iteratively improve an initial solution by local transformations, replacing a current solution by a better neighbor until no more improving moves are possible. In that case, a local optimum is reached and the procedure stops. Most well-known LS algorithms for the TSP are based on k-opt exchanges which delete k arcs of a current solution and reconnect partial tours with k other arcs in all possible ways. Figure 1 describes the specific 3-opt [2] procedure. Among the other popular ones, we may cite the 2-opt [3] and Lin-Kernighan [4] algorithms.

One of the key elements of LS is the pivoting rule which dictates the choice of the neighbor solution that will replace the current one [5]. The most commonly used methods are best-improvement and first-improvement. In the first case, all neighbors of the current solution are evaluated and the one which produces the greatest improvement is selected. In the second case, the first improving move is accepted and the others are discarded. Less popular alternatives are random-improvement and least-improvement which respectively choose a neighbor randomly or with minimal improvement of the objective function. Anderson [6] defines a parameter k associated to the number of improving moves found before choosing the best one. When $k = 1$, the algorithm uses a first-improvement strategy and

the more k increases, the more thorough is the exploration of the neighborhood. He deduces that first-improvement is generally the best choice for TSP.

In order to accelerate the execution of LS algorithms for the TSP, various mechanisms are usually used to reduce the neighborhood of the current solution. First, candidates lists, which comprise the cl nearest cities for each city, may be used when reconnecting partial tours instead of the whole set of cities. Second, fixed-radius neighbor search reconnects tours only with arcs for which the sum of weights is potentially lower than the sum of weights of the arcs to be deleted. Third, *don't look bits* are associated to each city in order to drive the search away from arcs which have recently led to unimproving moves. A complete description of these methods may be found in Bentley [7] and in Johnson and McGeoch [8].

A LS procedure becomes trapped in a local optimum when no improving moves are possible in the neighborhood of the current solution. A way to partly counter this problem is to embed it in a guiding construction such as Iterated Local Search (ILS) [9]. This metaheuristic is divided into four main steps that are highlighted in Figure 2. The first one is the generation of an initial solution S, usually with constructive heuristics or randomly. The second one is the LS procedure that is applied to S to bring it to a local optimum. The third one is a perturbation move which transforms S into S' in order to get it out of that local optimum. Finally, an acceptance criterion is evaluated to choose which solution between S and S' will be used to resume the search. The three last steps are then repeated until an end criterion is reached, for example a maximum time limit or iteration count.

```
while S is not a local optimum do
    for each combination of a, b, c ∈ [0; n] do
        Delete arcs (a,a+1), (b,b+1) and (c,c+1)
        Produce neighbors of S by reconnecting partial tours
        Evaluate neigbors of S
    Replace S by the best neighbor chosen by the pivoting rule
Return best solution S
```

Fig. 1: 3-opt LS pseudo-code.

```
Generate solution S
Apply LS procedure on S
Evaluate length L of solution S
while end criterion is not reached do
    Transform S into S' by a perturbation move
    Apply LS procedure on S'
    Evaluate length L' of solution S'
    if L' < L then replace S by S' // acceptance criterion
Return best solution S
```

Fig. 2: ILS pseudo-code.

The ILS metaheuristic is considered as one of the most powerful approximate methods for the TSP [1]. In fact, the works of Stützle and Hoos [10] and Lourenço *et al.* [9] show its competitiveness in solving various TSP problems varying from 100 to 5915 cities. However, faced to large and hard optimization problems, it may need a considerable amount of computing time and memory space to be effective in the exploration of the search space. A way to accelerate this exploration is to use parallel computing.

3. Literature review on parallel LS

Verhoeven and Aarts [11] proposed a classification that distinguishes *single-walk* and *multiple-walk* parallelization approaches for LS algorithms. In the first category, one search process goes through the search space and its steps are decomposed for parallel execution. In that case, neighbors of a solution may be evaluated in parallel (*single-step*) or several exchanges may be performed on different parts of that solution (*multiple-step*). In the second category, many search processes are distributed over processing elements and designed either as *multiple independent walks* or *multiple interacting walks.*

Johnson and McGeoch [8] defined three parallelization strategies for k-opt algorithms. The first one uses *geometric partitioning* to divide the set of cities into subgroups that are sent to different processors to be improved by a constructive algorithm and a LS procedure. As this partitioning has the drawback of isolating subgroups without reconnecting subtours intelligently, the second strategy favors *tour-based partitioning* to divide tours into partial solutions that includes a part of the edges of the current solution. The third approach is a simple parallelization of neighborhood construction and exploration.

Works on parallelization of ILS for the TSP mainly follow the population-based, multiple-walk approach where many solutions are built concurrently. Hong *et al.* [12] designed a parallel ILS which executes a total of m iterations using a pool of p solutions. Martin and Otto [13] proposed an implementation in which several solutions are computed simultaneously on different processors and the best solution replaces all solutions at irregular intervals.

Few authors have tackled the problem of parallelizing LS algorithms for TSP on GPU. Luong *et al.* [14], [15] proposed a methodology for implementing large neighborhood algorithms in which the CPU is in charge of LS processes and the GPU deals with the generation and evaluation of neighbor solutions. It was experimented on TSP with Tabu Search using a swap as the local transformation. Maximal speedup of 19.9 is reported on a 5915 cities problem but solution quality is not provided. O'Neil *et al.* [16] implemented an iterative hill climbing algorithm based on 2-opt local transformations in which random restarts are associated to threads. Maximal speedup of 61.9 is reported on a 100 cities problem. Fujimoto and Tsutsui [17] integrated a 2-opt best-improvement LS into a genetic algorithm executed

on GPU. Maximal speedup of 24.20 is obtained on problems of up to 493 cities. Delévacq *et al.* [18] augmented an Ant Colony Optimization algorithm with a 3-opt local search implemented on GPU. Solving TSP problems from 198 to 2103 cities, they reported speedups of up to 8.03 with solution quality similar to the best known sequential implementation.

These works provide a frame of reference for evaluating the attainable efficiency of GPU-based LS algorithms for the TSP. However, most of them overlook important issues that make it difficult to assess their effectiveness as parallel optimization methods. First, speedups are mostly provided with nonexistent or inappropriate evaluation of solution quality. The proposed implementations are also experimented on TSPs often limited to a few hundred cities without bypassing obvious memory limits of actual GPUs. In addition, as significant implementation details like the pivoting rule and the mechanisms used to accelerate the computations are not provided, algorithms can hardly be reproduced or tested on larger problems. Finally, basic local transformations are most often implemented even though the most effective methods for solving the TSP are based on the 3-opt neighborhood structure and the Lin-Kernighan algorithm [6].

As LS algorithms are key underlying components of most high-performing metaheuristics, a natural fit is to run a guiding metaheuristic on CPU while the GPU, acting as a co-processor, takes charge of running the LS procedure. However, there is still much conceptual, technical and comparative work to achieve in order to design such hybrid parallel combinatorial optimization methods. This paper aims to partially fill this gap by proposing, evaluating and comparing various parallelization strategies for the 3-opt LS on GPU.

4. GPU parallelization strategies for ILS

We propose three parallelization strategies based on the algorithm described in Figures 1 and 2. They mainly differ in the pivoting rule used to select an improving neighbor, the distribution of solutions to processing elements and the use of GPU shared memory. For the sake of completeness, we first provide a brief description of the GPU architecture and computational model.

The conventionnal NVIDIA GPU [19] architecture includes many *Streaming Multiprocessors* (SM), each one of them being composed of *Streaming Processors* (SP). On this special hardware, the *global* memory is a specific region of the *device* memory that can be accessed in read and write modes by all SPs of the GPU. It is relatively large in size but slow in access time. Each SM employs an architecture model called *SIMT* (*Single Instruction, Multiple Thread*) which allows the execution of many coordinated threads in a data-parallel fashion. *Constant* and *texture memory caches* provide faster access to device memory but are read-only. The constant memory is very limited in size whereas texture memory size can be adjusted in order to occupy the available device memory. All SPs can read and write in their *shared memory*, which is fast in access time but small in size and local to a SM. It is divided into memory banks of 32-bits words that can be accessed simultaneously. *Registers* are the fastest memories available on the GPU but involve the use of slow local memory when too many are used.

In the CUDA programming model [19], the GPU works as a co-processor of a conventional CPU. It is based on the concept of kernels, which are functions (written in C) executed in parallel by a given number of CUDA threads. These threads are grouped together into *blocks* that are distributed on the GPU SMs to be executed independently of each other. However, the number of blocks that a SM can process at the same time (*active blocks*) is restricted and depends on the quantity of registers and shared memory used by the threads of each block. In a block, the system groups threads (typically 32) into *warps* which are executed simultaneously on successive clock cycles. The number of threads per block must be a multiple of its size to maximize efficiency. Much of the global memory latency can then be hidden by the thread scheduler if there are sufficient independent arithmetic instructions that can be issued while waiting for the global memory access to complete. Consequently, the more active blocks there are per SM, and also active warps, the more the latency can be hidden. Special care must also be taken to avoid flow control instructions (if, switch, do, for, while) that may force threads of a same block to take different paths in the program and serialize the execution.

The proposed strategies are inspired by the population-based and multiple independent walks general strategies described in Section 3. However, only the LS phase is parallelized on GPU instead of entire walks. In all cases, memory management issues had to be adressed. Data transfers between the CPU and the GPU as well as global memory accesses require considerable time but may be often be reduced by storing the related data structures in shared memory. However, in the case of ILS applied to TSP, one central data structure is the distance matrix which is needed by all solutions of the population while being too large ($O(n^2)$ in size) to fit in shared memory for problems larger than a few hundred cities. It is then kept in global memory. On the other hand, as it is not modified during the LS phase, it is possible to take benefit of the texture cache to reduce their access times. The first two strategies are applied on a first-improvement LS implementation and the last one on a best-improvement scheme over a fixed size neighborhood. Specific details for each strategy are given in the next sections.

4.1 First-improvement LS : $ILS-FI_{thread}$ and $ILS-FI_{block}$

The proposed GPU parallelization strategies applied to a first-improvement LS have been presented by the authors

in the form of preliminary work [20] and are inspired by our previous contributions on Ant Colony Optimization [21], [18]. Their general parallelization model is illustrated in Figure 3(a). The first one, presented in Figure 3(b), is called $ILS-FI_{thread}$ and associates each LS to a CUDA thread. It has the advantage of allowing the execution of a great number of LSs on each SM and the drawback of limiting the use of fast GPU memory. The second strategy, called $ILS-FI_{block}$ and illustrated in Figure 3(c), associates each LS to a CUDA block. Thus, parallelism is preserved for the LS phase, but another level of parallelism is exploited by sharing the multiple neighbors between many threads of a block. As only one solution is assigned to a block, it becomes possible to store the data structures needed to improve the solution in the shared-memory. Two variants of the $ILS-FI_{block}$ strategy are then distinguished : $ILS-FI_{block}^{global}$ and $ILS-FI_{block}^{shared}$.

Currently, most efficient LS methods for solving the TSP are based on the first-improvement pivoting rule and mechanisms to reduce execution times. They were obviously designed and optimized for traditional single processor computing systems. While it may be possible to achieve good efficiency on small multiple processor/core systems with minimal changes to existing algorithms, the parallelization process is not as straightforward for a synchronous, massively parallel architecture like the GPU. On one hand, existing sequential implementations are based on reducing the number of computed neighbors in a single LS step whereas GPU processing is well suited for large data-parallel applications. On the other hand, speedup mechanims are often based on conditional statements which induce thread divergence within warps and then serialization. These observations lead us to propose a new parallelization strategy based on the synchronous evaluation of fixed size neighborhoods.

4.2 Best-improvement LS : $ILS-RKBI_{blocks}$

In a basic best-improvement LS, the whole neighborhood of a given solution is evaluated. In the case of the 3-opt move for a n cities TSP, the number of combinations to delete three edges of the tour is $\binom{n}{3}$. For each combination, there are 7 ways to reconnect the subtours [22] which corresponds to 7 neighbors. Research on TSP show that such a scheme leads to prohibitive execution times and local optima that are much further away from the ones obtained with first-improvement schemes [6]. We then propose the *random-k-best-improvement* pivoting rule which offers a compromise between first-improvement and best-improvement as well as a computation model better suited to GPUs. At each step, arcs to be deleted are randomly selected to generate a total of k neighbors and the one which produces the greatest improvement is kept. By assigning different values to k, one may customize the search behavior and the amount of work performed by the GPU. This leads to the proposition of a third parallelization strategy called $ILS-RKBI_{blocks}$ and illustrated in Figure 4(a).

This strategy splits computations of a single step into two kernels. The first one, described in Figure 4(b), is dedicated to neighborhood evaluation which is the most expensive part of the step. Each solution is associated to several blocks and its neighbors are splitted into groups to be assigned

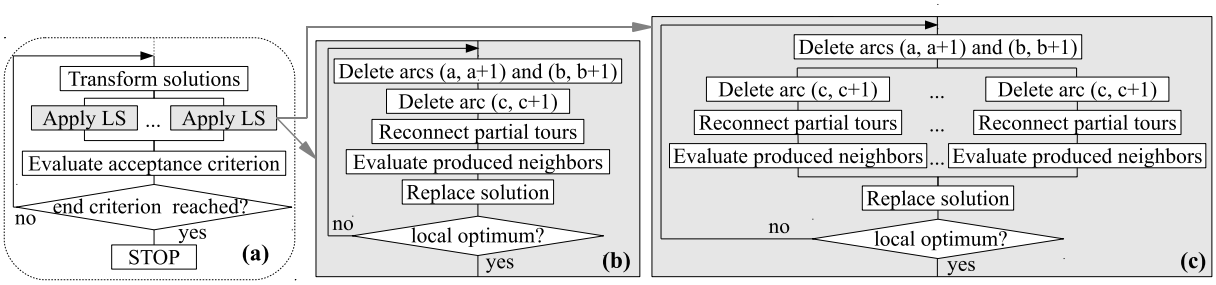


Fig. 3: Parallelization models for first-improvement ILS : general (a), $ILS-FI_{thread}$ (b) and $ILS-FI_{block}$ (c).

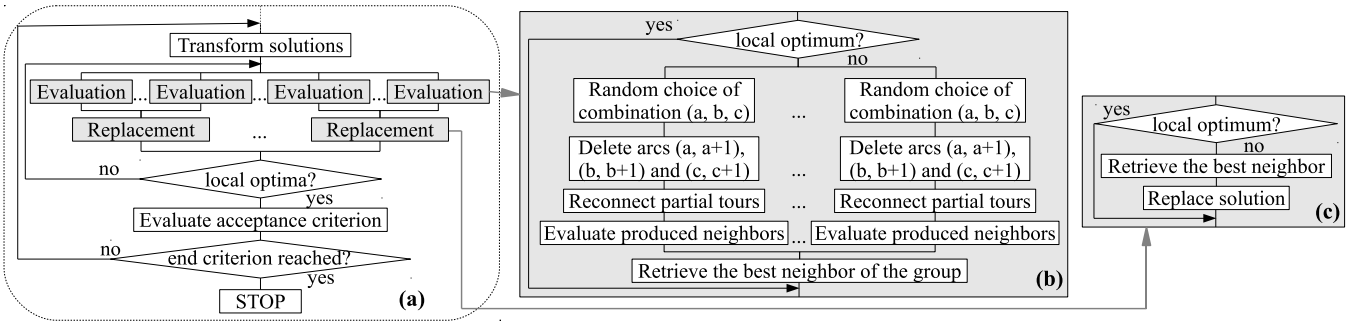


Fig. 4: Parallelization model for random-k-best-improvement ILS (a), evaluation kernel (b) and replacement kernel (c).

to each block. Each thread is then associated to several neighbors. Using the NVIDIA CURAND library, a thread randomly selects the three arcs to break, evaluates possible reconnections and stores the best one in its own space in shared memory. Once all the neighbors have been evaluated, a reduction is performed in the shared memory to find the best neighbor of the block which is stored in global memory.

In the second kernel, presented in Figure 4(c), each solution is associated to a single block which retrieves its best neighbor among the ones produced by the first kernel and replaces the current solution. The two kernels are launched alternately until all solutions are local optima.

5. Experimental results

GPU strategies are experimented on TSP problems ranging in size from 100 to 3038 cities. Speedups are computed by dividing the sequential CPU time with the parallel time obtained with the GPU acting as a co-processor. Experiments were made on a NVIDIA Fermi C2050 GPU containing 14 SMs, 32 SPs per SM and 48 KB of shared memory per SM. Code was written in the "C for CUDA V4.1" [19] programming environment. As a premiminary step, we validated our sequential ILS implementation with a comparative study with Stützle and Hoos [10] and Lourenço *et al.* [9] works.

The parallel ILS parameters are as follows. Initial solutions are built with the nearest neighbor heuristic, improved with 3-opt LS and perturbed with a double-bridge move. A population of nb_{sol} solutions is used, a total number of it_{total} iterations is performed and the ILS procedure is limited to $it_{lim} = \frac{it_{total}}{nb_{sol}}$ iterations for each solution. All speedups are computed for $nb_{sol} = 2^x$ with $x \in \{0, 3, 6, 9, 11\}$ from 20 trials for problems with less than 1000 cities and from 10 trials for larger instances.

5.1 $ILS - FI_{thread}$ and $ILS - FI_{block}$

First-improvement LS algorithms use a *don't look bits* procedure and a fixed-radius neighbor search restricted to candidate lists of size 40. it_{total} is set to 1048576 for each problem to ensure that the same number of LSs are performed for all numbers of blocks and threads. In $ILS - FI_{block}$, the number of blocks is set to nb_{sol} and the number of threads per block is set to the size of candidate lists. In $ILS - FI_{thread}$, the number of blocks and threads are configured to maximize the number of blocks without exceeding the number of active blocks per SM.

Table 1 shows speedup for $ILS - FI_{thread}$, $ILS - FI^{global}_{block}$ and $ILS - FI^{shared}_{block}$. The reader may note that increasing nb_{sol} and so, the total number of threads, leads to increasing speedups for all strategies. Overall, if the number of threads used is too small, GPU resources are not well exploited and memory latency is not efficiently hidden. The reader may also notice that speedups obtained with $ILS - FI_{thread}$ are always lower than with $ILS - FI_{block}$. As this strategy does not execute enough threads in parallel to efficiently hide memory latency, we obtain a maximal speedup of 2.02. In fact, speedups are achieved only with 2048 threads. Furthermore, code divergence induced by computing the neighbors of many solutions/threads on the same block in SIMD mode involves significant algorithm serialization.

Table 1: Speedup for $ILS - FI_{thread}$ (T), $ILS - FI^{global}_{block}$ (BG) and $ILS - FI^{shared}_{block}$ (BS) strategies and solution quality (frequency of finding the known optimum $freq$ and average percentage deviation from the optimum Δ_{avg}) for each problem.

Problem	nb_{sol}	Speedup			Solution quality	
		(T)	(BG)	(BS)	$freq$	Δ_{avg}
kroA100	1	0.01	0.08	0.08	1.00	0.000
	8	0.06	0.45	0.50	1.00	0.000
	64	0.34	2.83	3.13	1.00	0.000
	512	0.93	6.40	7.01	1.00	0.000
	2048	2.02	7.07	**7.83**	1.00	0.000
lin318	1	0.01	0.09	0.10	1.00	0.000
	8	0.06	0.51	0.57	1.00	0.000
	64	0.35	3.23	3.54	1.00	0.000
	512	0.84	6.92	7.61	1.00	0.000
	2048	1.72	7.77	**8.51**	1.00	0.000
rat783	1	0.01	0.07	0.08	1.00	0.000
	8	0.06	0.43	0.49	1.00	0.000
	64	0.34	2.69	2.18	1.00	0.000
	512	0.80	5.77	3.13	0.23	0.020
	2048	1.49	**6.37**	3.27	0.00	0.145
fl1577	1	0.01	0.07	0.08	0.33	0.182
	8	0.05	0.33	0.40	0.39	0.010
	64	0.23	1.73	0.97	0.67	0.003
	512	0.54	4.59	1.14	0.10	0.015
	2048	0.79	**5.20**	1.13	0.00	0.074
pcb3038	1	0.01	0.08	-	0.00	0.210
	8	0.06	0.46	-	0.00	0.192
	64	0.39	2.91	-	0.00	0.306
	512	0.76	5.20	-	0.00	0.667
	2048	1.27	**5.60**	-	0.00	1.104

The greater speedups and the maximal value of 7.77 obtained with $ILS - FI^{global}_{block}$ show that sharing the work associated to each solution between several threads is more efficient. For example, when nb_{sol} is set to 2048, ILS_{thread} uses 2048 threads versus 81920 for ILS_{block}. On the other hand, speedups increase from 100 to 318 cities and then slightly decrease. In that case, the larger data structures and frequent memory accesses needed to solve the biggest problems imply memory latencies that grow faster than the benefits of parallelizing available computations. Further improvements are brought by the use of shared memory, introduced in $ILS - FI^{shared}_{block}$, which provides maximal speedup of 8.51. However, results for the three biggest problems show that the limits of this kind of memory are quicky reached. In fact, as it is limited in size, using too much of it reduces the number of active blocks per SM. Associated to the combined effect of the increasing number of blocks required to perform computations, performance

gains become less significative. For the 3038 cities problem, the amount of required shared memory is so high that no block can be active on any SM.

An analysis of the frequency of finding the known optimum and the average percentage deviation from the optimum is also provided in Table 1. It shows that the optimal solution is always found by the parallel implementations for small problems. For medium-sized problems, the more nb_{sol} increases, the less frequently is the optimal solution found. As the number of iterations becomes too low to provide a thorough search, the optimal solution is never found for the bigger problem. This indicates that when choosing appropriate parameters for the parallel algorithms, a compromise must be achieved between speedup and solution quality.

5.2 $ILS - RKBI_{blocks}$ strategy

An empirical study was performed to determine the parameters used for the $ILS-RKBI_{blocks}$ strategy. The number of threads per block is set to 64 as it generally maximizes the number of active blocks per SM. It is also a multiple of 32 as advised in the CUDA specification [19]. Table 2 provides the chosen number of blocks for each solution to evaluate its neighbors. The number of iterations it_{total} and the number of evaluated neighbors k are selected so the sequential execution time is in the same order of magnitude as the ones of the first-improvement strategies when nb_{sol} is set to 64. They are then set to $1,120,000/5,500$ for kroA100, $1,120,000/11,000$ for lin318, $560,000/10,000$ for rat783, $2,240,000/1,200$ for fl1577 and $2,240,000/1,000$ for pcb3038.

Table 2: Number of blocks per solution for each problem and nb_{sol} values.

	1	8	64	512	2048
kroA100	120	70	30	5	2
lin318	110	60	30	5	2
rat783	80	100	60	10	5
fl1577	140	90	60	15	15
pcb3038	120	100	90	20	20

Table 3 presents speedup, frequency of finding the optimum and average percentage deviation from the optimum for the $ILS-RKBI_{blocks}$ strategy. The reader may notice that significant speedups are obtained with all values of nb_{sol} and a maximal speedup 45.40 is achieved with 64 solutions on the 318 cities problem. Also, with the exception of the two smallest problems with $nb_{sol} = 1$ where the amount of work is too small, speedups are in the same order of magnitude with any value of nb_{sol} for any particular problem. This shows the scalability of the neighborhood distribution strategy to different population sizes. However, the more nb_{sol} increases, greater is the deterioration of solution quality in most cases.

Table 3: Speedup and solution quality (frequency of finding the known optimum $freq$ and average percentage deviation from the optimum Δ_{avg}) for ILS_{rkbi}.

Problem	nb_{sol}	Speedup	Solution quality	
			$freq$	Δ_{avg}
kroA100	1	25.06	1.00	0.000
	8	37.21	1.00	0.000
	64	**45.29**	1.00	0.000
	512	42.12	1.00	0.000
	2048	42.32	1.00	0.000
lin318	1	27.55	0.00	0.482
	8	38.35	0.00	0.483
	64	**45.40**	0.00	0.633
	512	43.21	0.00	0.875
	2048	43.58	0.00	0.886
rat783	1	14.76	0.00	1.224
	8	16.74	0.00	1.582
	64	**17.02**	0.00	2.740
	512	16.24	0.00	3.496
	2048	16.74	0.00	3.518
fl1577	1	**30.40**	0.00	1.034
	8	29.47	0.00	1.703
	64	29.52	0.00	1.831
	512	28.02	0.00	1.776
	2048	28.85	0.00	1.741
pcb3038	1	31.23	0.00	4.636
	8	31.20	0.00	4.940
	64	31.70	0.00	5.162
	512	30.31	0.00	5.104
	2048	**33.59**	0.00	4.970

A comparison with Table 1 shows that when speedup is considered, $ILS-RKBI_{blocks}$ clearly performs better than $ILS - FI$ in all cases. The increased neighborhood size, better sharing of neighbors between blocks and reduction of thread divergence allows the resulting algorithm to make a better use of GPU ressources. Using fixed size data structures to store the best neighbors also makes the use of shared memory relevant even for bigger problems. However, the gains in speedup also lead to the deterioration of solution quality. The random choice of neighbors and inhability to use speedup mechanisms of first-improvement strategies makes it difficult for $ILS - RKBI_{blocks}$ to keep the same level of optimization than $ILS - FI_{thread}$ and $ILS - FI_{block}$ in the same execution time. This may indicate either that a compromise must be found between speedup and solution quality when designing parallel ILS algorithms for GPUs or that news ways must be thought to restore - if possible - the delicate balance between all the algorithm components in this context.

6. Conclusion

The aim of this paper was to design efficient parallelization strategies for Iterated Local Search on Graphics Processing Units to solve the Travelling Salesman Problem. The $ILS-FI_{thread}$ and $ILS-FI_{block}$ strategies associated the local search phase to the execution of streaming processors and multiprocessors respectively. They provided maximal

speedups of 2.02 and 8.51 with competitive solution quality as well as major shortcomings in their use of GPU computational resources. In an attempt to overcome these limitations, the $ILS - RKBI_{blocks}$ strategy was proposed to increase neighborhood size and associate multiple blocks to the evaluation of each solution. Significant speedups were then achieved, ranging as high as 45.40, at the cost of a variable deterioration of solution quality. This leads to the idea that solving the TSP and other combinatorial optimization problems on GPU is currently a matter of compromise between speedup and search efficiency.

In future works, we plan to deepen our understanding of the links between neighborhood size, pivoting rules and algorithm parameters in order to improve the search process of our GPU ILS implementation. Also, as this work is part of a greater project related to the parallelization of metaheuristics on GPU, we seek to use the knowledge built in this paper to propose a general framework that can be applied to other metaheuristics. We also plan to propose algorithms to automatically determine effective thread/block/GPU configurations for ILS and other metaheuristics. We believe that the global acceptance of GPUs as components for optimization systems requires algorithms and software that are not only effective, but also usable by a wide range of academicians and practitioners.

Acknowledgment

This work has been supported by the Agence Nationale de la Recherche (ANR) under grant no. ANR-2010-COSI-003-03. The authors would also like to thank the Centre de Calcul Régional Champagne-Ardenne for the availability of the computational resources used for experiments.

References

[1] H. Hoos and T. Stützle, *Stochastic Local Search Foundations and Applications*. Morgan Kaufmann, Elsevier, 2004.

[2] S. Lin, "Computer solutions of the traveling salesman problem," *Bell System Technical Journal*, vol. 44, pp. 2245–2269, 1965.

[3] G. A. Croes, "A method for solving traveling salesman problems," *Operations Research*, vol. 6, pp. 791–812, 1958.

[4] S. Lin and B. Kernighan, "An effective heuristic algorithm for the travelling salesman problem," *Operations Research*, vol. 21, pp. 498–516, 1973.

[5] M. Yannakakis, "The analysis of local search problems and their heuristics," in *STACS*, ser. Lecture Notes in Computer Science, vol. 415. Springer, 1990, pp. 298–311.

[6] E. Anderson, "Mechanisms for local search," *European Journal of Operational Research*, vol. 88, no. 1, pp. 139–151, 1996.

[7] J. Bentley, "Fast algorithms for geometric traveling salesman problems," *ORSA Journal on Computing*, vol. 4, no. 4, pp. 387–411, 1992.

[8] D. Johnson and L. McGeoch, *The Travelling Salesman Problem: A Case Study in Local Optimization*, ser. E.H.L. Aarts and J.K. Lenstra, editors, Local Search in Combinatorial Optimization. John Wiley & Sons, 1997, pp. 215–310.

[9] H. Lourenço, O. Martin, and T. Stützle, *Iterated local search: framework and applications*, ser. Handbook of metaheuristics. Springer, 2010, pp. 363–397.

[10] T. Stützle and H. Hoos, *Analysing the run-time behaviour of iterated local search for the traveling salesman problem*, ser. Essays and surveys in metaheuristics. Springer, 2001, pp. 21–43.

[11] M. Verhoeven and E. Aarts, "Parallel local search," *Journal of Heuristics*, vol. 1, pp. 43–65, 1995.

[12] I. Hong, A. Kahng, and B. Moon, "Improved large-step markov chain variants for the symmetric tsp," *Journal of Heuristics*, vol. 3, pp. 63–81, September 1997.

[13] O. Martin and S. Otto, "Combining simulated annealing with local search heuristics," *Annals of Operations Research*, vol. 63, pp. 57–75, 1996.

[14] T. Luong, N. Melab, and E. Talbi, "Neighborhood structures for gpu-based local search algorithms," *Parallel Processing Letters*, vol. 20, no. 4, pp. 307–324, 2010.

[15] ——, "Gpu computing for parallel local search metaheuristic algorithms," *IEEE Transactions on Computers*, vol. 99, no. PrePrints, 2011.

[16] M. O'Neil, D. Tamir, and M. Burtscher, "A parallel gpu version of the traveling salesman problem," in *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2011, pp. 348–353.

[17] N. Fujimoto and S. Tsutsui, "A highly-parallel tsp solver for a gpu computing platform," in *NMA*, 2010, pp. 264–271.

[18] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki, "Parallel ant colony optimization on graphics processing units," *Journal of Parallel and Distributed Computing*, 2012.

[19] *CUDA : Computer Unified Device Architecture Programming Guide 4.1*, 2012. [Online]. Available: http://www.nvidia.com

[20] A. Delévacq, P. Delisle, and M. Krajecki, "Parallel gpu implementation of iterated local search for the travelling salesman problem," in *Learning and Intelligent OptimizatioN*. Lecture Notes in Computer Science, 2012.

[21] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki, "Parallel ant colony optimization on graphics processing units," in *Proceedings of the 16th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*, H. R. Arabnia, S. C. Chiu, G. A. Gravvanis, M. Ito, K. Joe, H. Nishikawa, and A. M. G. Solo, Eds. CSREA Press, 2010, pp. 196–202.

[22] G. Gutin and A. Punnen, *The Traveling Salesman Problem and Its Variations*, ser. Combinatorial Optimization. Kluwer Academic Publishers, 2002.

Evaluation of power consumption in programming models based on map reduce in shared memory systems

Zahra khoshmanesh[1]
[1]CSE and IT Department, Shiraz University, Shiraz, Iran

Abstract *- One of the most common models in parallel programming is Map reduce. In companies which use Map reduce framework, at each time a lot of computers are executing Map and Reduce functions. These functions are executing many times and we can estimate the effect of any small changes in response speed or consumed power in each execution of the Map reduce model to be very high. On the other hand, power and energy became an important challenge in computer systems with high performance, so that criteria such as consumed Power are as important as performance criteria. Nowadays, because of the generated heat and also because of decrease in energy sources, saving the consumed power is very important. So, finding the parts of the programs which needs more power is of prime Importance, because by finding these parts, we can find the ways to investigate and improves them in terms of power. In this paper, we investigate the available Map Reduce framework and programs in this regard from the point of consumed power, which are implemented in multi-core environment with common memory.*

Keywords: *consumed power, map reduce, multicores, parallel programs, efficiency*

1 Introduction

According to Moore law [5], numbers of transistors on a chip are doubled each two years. As the size of the transistors is reduced, more of them can be placed on the chips, so we can have more cores on a chip. But the problem which arises is the consumed power. One transistor has a low consumed power by itself, but here we have a big number of transistors which cause high amount of the heat. On the other hand, transistors cannot be turned off completely, this means even if they are completely turned off, they will have current leakage, and some currents will pass through them, even when they are turned off. And this will cause gradual loose of energy and the power of connection wires will be too high. Most power saving mechanisms like doing the task slowly, will affect the performance. For two reasons, increasing the speed of the processors by increasing the frequency is not possible. These two challenges are memory wall and heat wall. Memory wall is related to difference in speed between memory and main processor and heat wall related to the fact that as the execution speed of the computer increases, consumed power which is proportional to cubic root of the frequency, also will increase and subsequently more heat will be generated [7]. To overcome these problems and to increase computation power of the system, parallel architecture was recommended among which multi-core architecture was designed as the practical way of overcoming problems. So, we have a number of cores on a processor and in order to be able to use the power of all cores, we should use multi-node programming. But the problem which arises is that common multi-node programming leaves the whole control to the user, and this is a disadvantage.

By increasing the use of information technology and popularity of the issues such as automation and use of digital equipment in business processes, in all cases we face data generation, nowadays we come across with the problem named data volume explosion. For example EBay Company has announced that, it has more than 6.5 petabyte data. This figure is 10 petabyte for Yahoo. The need to analyze the raw data for different uses in increasing which in turn demand appropriate and effective solutions for data analysis.

In 2004, Google introduced its programming model for use in environment with several processor units which is called Map reduce. This model which is inspired from functional programming model does all the operations related to passing, distributing and gathering data between computers and just demands the computing core of the program from user. In today world, in which we deal with a high amount of the data, this programming model is very useful. If we consider each computer in a distributed system as a core in multi-core processors, we come with the conclusion that models such as Map reduce is a good choice for use in this processing unit.

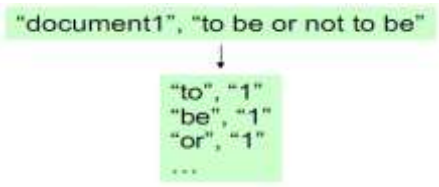
This model is a simple programming model which is used for solving computation problems in big scales and in distributing form. Map reduce was developed by Google in 2003 and is a software framework which provide a safe and scalable bed for development of distributing uses and is implemented in different languages.

In fact, it contains a set of library functions which hide the details and sophistication from the user. These details include: automatic paralleling of the tasks, data load balancing, optimization of network and disk transferring, management of faults in machines. Moreover, each improvement in library will be applied to all the places

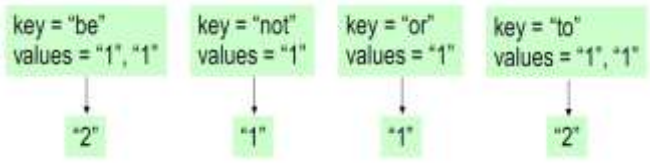
which this library has been used.In this method, two main steps exist: Map and Reduce.

Map step: main node, take the input and divides it into smaller sub-problems and then distribute them between nodes which are responsible for doing the tasks. It is possible that, this node repeats the task and if so, we would have a multi-surface architecture. Finally, these sub-problems are processed and response is sent to the output.

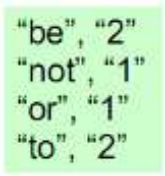
Reduce step: for generating an output, the responses and results which are received by main node, will be merged together. To do so, some operations like filtering and conversion may be applied on the data.
These two main operations are done on a regular pair (key, value). Map function, take a regular pair of the data and convert it to a list of regular pairs. Then, Map reduce framework, gather all the pairs with the same keys from all lists and produce a group. For each generated key, one group is produced. And the reduce function act on all groups. Now, map reduce framework, convert one list of (key, value) to a list of values.
As an example, a framework called Mars is designed for graphical processors for programs based on Map reduce. Also some programs which are designed and developed by Google based on the map reduce and focus on web based search programs for ordinary CPU are tested and implemented GPUs with high computation power and broad band widths.
One of the common examples for solving problems by Map reduce, is finding the number of occurrence of a word in a document. Here document referee to web page. In this problem, input is a file which has a text in each row. Map function takes (key, value) pairs. In this case, key is the address of the web page, and value is web content .[8] Output of the map function will be a list of other regular pairs :(number of occurrence, word) same as figure below:



Now map reduce framework gathers all the pairs with common keys. Then reduce function, merges the value of the pairs with common keys and assign a new value for that which in this case is sum of the values.



And finally, output will be like this:



Word count pseudo codes are shown in following:

```
map(String key, String value):
 // key: document name
 // value: document contents
 for each word w in value:
  EmitIntermediate(w, "1");
reduce(String key, Iterator values):
 // key: a word
 // values: a list of counts
 int result = 0;
 for each v in values:
  result += ParseInt(v);
 Emit(AsString(result));
```

Algorithm 1 .word count pseudo-code

Studies show that direct monthly energy costs and expenses for data center is about 23% of the total monthly operational costs [9]. If we consider costs like power distribution and cooling structure which affect the monthly energy costs indirectly, it will be about 42% of monthly operational costs. Trends show that performance of the processors increase each 18 month in terms of number of cores, while performance is doubled in each wat in each two years. [10]. so, it would not be surprising if a previous study has estimated that servers in USA include 3% of the total consumed energy in 2011. One reason for the high cost of the energy of the servers is that nodes in clusters environment are used 20 to 30% and the performance of the energy in this range is below 50%. This reveals the fact that 42% of monthly operational cost contribute to power, which decrease in it will increases the energy performance. [11]
Since map reduce framework can be used in computers in a data center and its power can be determinable, investigation and analysis of power is necessary in these environments. [12, 13]

2. Previous works

In this part a brief description of the works which have been done before, is presented. In [12] Map reduce programming model for systems with common memory called Phoenix is implemented. Creation of the nodes, dividing the data, dynamic work timing and fault tolerance between nodes of the processors are done automatically by Phoenix. In this paper, codes written by low level APIs such as P-thread were compared with those written by Map Reduce. Conclusion

was that performance of the Map Reduce programming model on systems with common memory is as good as simpler parallel codes. Despite, run time overloads, Phoenix yield the same performance results for most of the applicable programs. Obviously, there still exist programs which give better results in P-thread than in Map Reduce model.

In [8] a framework called Mars is designed for graphical processors for Map reduce based programs. Also some programs are designed and developed by Google based on the map reduce which focus on web based search programs for ordinary CPU. Are tested and implemented in this framework and for CPUs with higher computation power and higher band widths and then are compared with Phoenix which is a modern and updated Map Reduce framework on multi-core CPUs. Above mentioned framework hides the sophistications of GPU programming with map reduce interface. And finally, they came with 16 time faster execution of 6 common web programs compare to executions on a CPU with four cores.

In [14] is focused on power and energy for clusters which use Map Reduce programming model and propose techniques to decrease the consumed energy. This technique is turning off the nodes and attention goes towards the number of nodes to be chosen to be off and have direct impact on the consumed energy. Majority of the works which have been done in this paper is systematic consideration of different strategies for turning off the nodes in Map Reduce model and their impacts on total consumed energy and workloads response time. Two methods investigated are CS & AIS. In the first method some of the nodes with lower loads become off in low load period and the second method is turning off all the nodes in low use period which is proposed by the authors. These two methods are compared to each other and conclusion is that the second method which is proposed by them, give better results in both saving consumed energy and قresponse time which is shown by analytical models and laboratory results.

It is worth noting that in all the papers mentioned above just the performance aspect is taken into account and consumed energy and power are not considered at all and in [14] just the consumed power of Map Reduce programs in distributed environment is investigated. None of the papers dealt with consumed power of Map Reduce base programs in environments with common memory

3. Laboratory results

In this part first a brief explanation will be given about the framework in which the test is done and then the measurement results in multi-core environment with common memory related to Phoenix Map reduce will be presented.

3.1 The Phoenix System

Phoenix implements Map Reduce for shared-memory systems. Its goal is to support efficient execution on multiple cores without burdening the programmer with concurrency management. Phoenix consists of a simple API that is visible to application programmers and an efficient runtime that handles parallelization, resource management, and fault recovery. [12]

The current Phoenix implementation provides an application-programmer interface (API) for C and C++. The API includes two sets of functions. The first set is provided by Phoenix and is used by the programmer's application code to initialize the system and emit output pairs. The second set includes the functions that the programmer defines .Apart from the Map and Reduce functions; the user provides functions that partition the data before each step and a function that implements key comparison. The API is quite small compared to other models. The API is type agnostic. The function arguments are declared as void pointers wherever possible to provide flexibility in their declaration and fast use without conversion overhead.

The API guarantees that within a partition of the intermediate output, the pairs will be processed in key order. This makes it easier to produce a sorted final output which is often desired. There is no guarantee in the processing order of the original input during the Map stage.

3.2 Basic Operation and Control Flow

Figure 1 shows the basic data flow for the runtime system. The runtime is controlled by the scheduler, which is initiated by user code. The scheduler creates and manages the threads that run all Map and Reduce tasks. It also manages the buffers used for task communication. The programmer provides the scheduler with all the required data and function pointers through the scheduler args t structure.

After initialization, the scheduler determines the number of cores to use for this computation. For each core, it spawns a worker thread that is dynamically assigned some number of Map and Reduce tasks.

To start the Map stage, the scheduler uses the Splitter to divide input pairs into equally sized units to be processed by the Map tasks. The Splitter is called once per Map task and returns a pointer to the data the Map task will process.

The Map tasks are allocated dynamically to workers and each one emits intermediate <key, value> pairs. The Partition function splits the intermediate pairs into units for the Reduce tasks. The function ensures all values of the same key go to the same unit. Within each buffer, values are ordered by key to assist with the final sorting. At this point, the Map stage is over. The scheduler must wait for all Map tasks to complete before initiating the Reduce stage. [12]

Reduce tasks are also assigned to workers dynamically, similar to Map tasks. The one difference is that, while with Map tasks we have complete freedom in distributing pairs across tasks; with Reduce we must process all values for the same key in one task. Hence, the Reduce stage may exhibit higher imbalance across workers and dynamic scheduling is more important. The output of each Reduce task is already

sorted by key. As the last step, the final output from all tasks is merged into a single buffer, sorted by keys. The merging takes place in log2 (P/2) steps, where P is the number of workers used. While one can imagine cases where the output pairs do not have to be ordered, our current implementation always sorts the final output as it is also the case in Google's implementation [8].

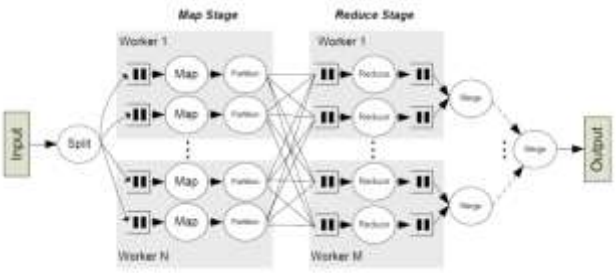


Fig. 1 the basic data flow for the phoenix runtime

For power consumer measurement, we use a power measurement device, applications of study with a brief description of them include:

Word Count: It counts the frequency of occurrence for each word in a set of files. The Map tasks process different sections of the input files and return intermediate data that consist of a word (key) and a value of 1 to indicate that the word was found. The Reduce tasks add up the values for each word (key).
Reverse Index: It traverses a set of HTML files, extracts all links, and compiles an index from links to files. Each Map task parses a collection of HTML files. For each link it finds, it outputs an intermediate pair with the link as the key and the file info as the value. The Reduce task combines all files referencing the same link into a single linked-list.

Matrix Multiply: Each Map task computes the results for a set of rows of the output matrix and returns the (x,y) location of each element as the key and the result of the computation as the value. The Reduce task is just the Identity function.

String Match: It processes two files: the "encrypt" file contains a set of encrypted words and a "keys" file contains a list of non-encrypted words. The goal is to encrypt the words in the "keys" file to determine which words were originally encrypted to generate the "encrypt file". Each Map task parses a portion of the "keys" file and returns a word in the "keys" file as the key and a flag to indicate whether it was a match as the value. The reduce task is just the identity function
.

KMeans: It implements the popular kmeans algorithm that groups a set of input data points into clusters. Since it is iterative, the Phoenix scheduler is called multiple times until it converges. In each iteration, the Map task takes in the existing mean vectors and a subset of the data points. It finds the distance between each point and each mean and assigns the point to the closest cluster. For each point, it emits the cluster id as the key and the data vector as the value. The Reduce task gathers all points with the same cluster-id, and finds their centroid (mean vector). It emits the cluster id as the key and the mean vector as the value.

PCA: It performs a portion of the Principal Component Analysis algorithm in order to find the mean vector and the covariance matrix of a set of data points. The data is presented in a matrix as a collection of column vectors. The algorithm uses two Map Reduce iterations. To find the mean, each Map task in the first iteration computes the mean for a set of rows and emits the row numbers as the keys, and the means as the values. In the second iteration, the Map task is assigned to compute a few elements in the required covariance matrix, and is provided with the data required to calculate the value of those elements. It emits the element row and column numbers as the key, and the covariance as the value. The Reduce task is the identity in both iterations.

Histogram: It analyzes a given bitmap image to compute the frequency of occurrence of a value in the 0-255 range for the RGB components of the pixels. The algorithm assigns different portions of the image to different Map tasks, which parse the image and insert the frequency of component occurrences into arrays. The reduce tasks sum up these numbers across all the portions.

Linear Regression: It computes the line that best fits a given set of coordinates in an input file. The algorithm assigns different portions of the file to different map tasks, which compute certain summary statistics like the sum of squares. The reduce tasks compute these statistics across the entire data set in order to finally determine the best fit line.[12]

3.3 Measurement of consumed power of the programs

In this part each program with different set of the data (small, medium or large) have been executed and the consumed power of the different phases including splitters, map, reduce, partition, sort and hash is measured, if available, and the results are shown in the following graphs.

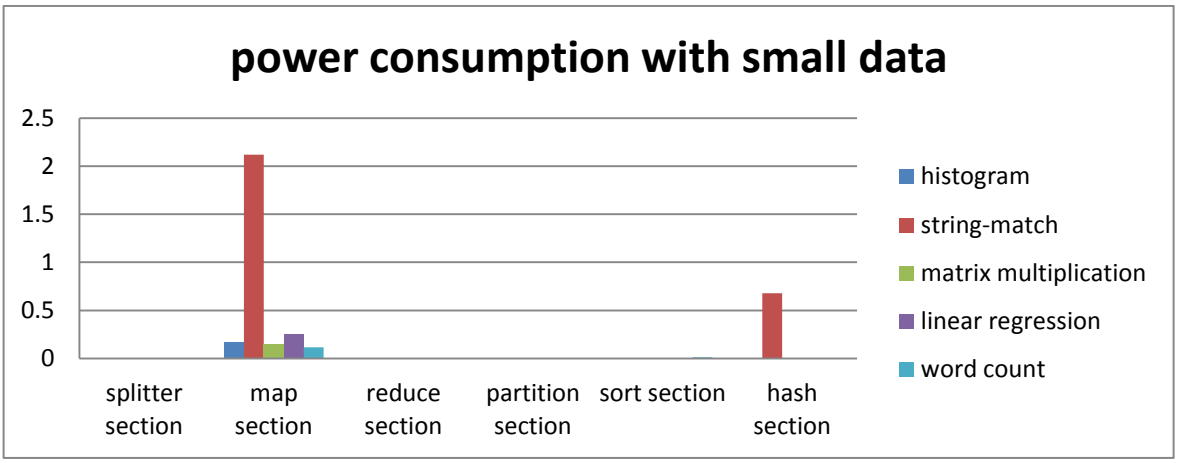


Fig .2 power consumption in small data

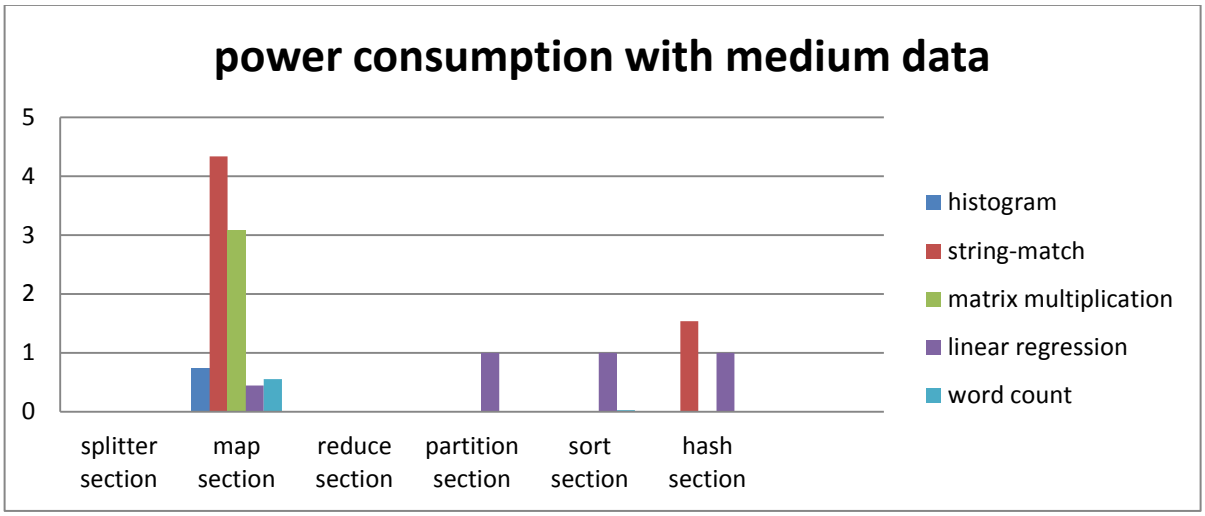


Fig .3 power consumption with medium data

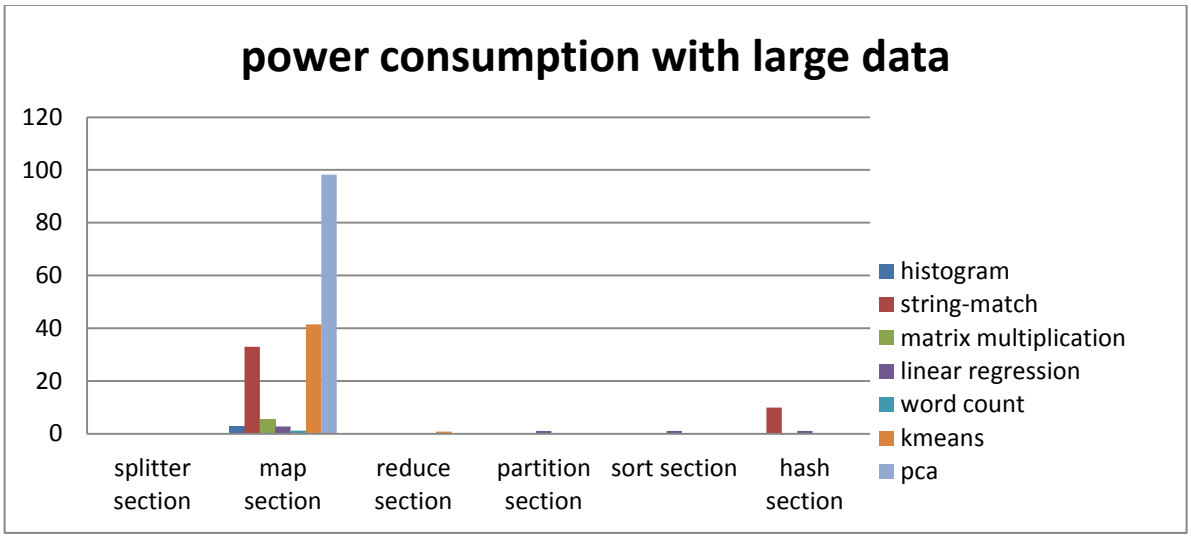


Fig .4 power consumption with large data

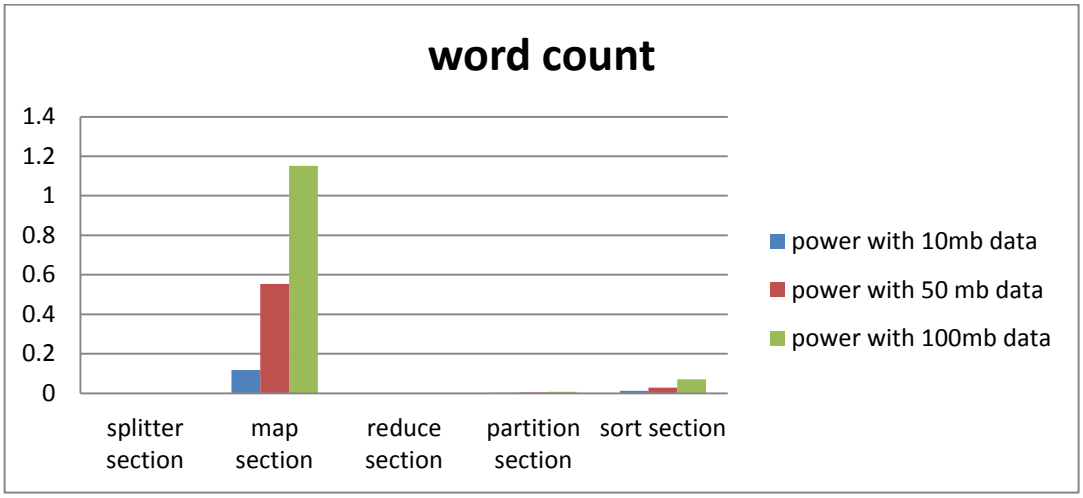


Fig.5 power consumption in word count application

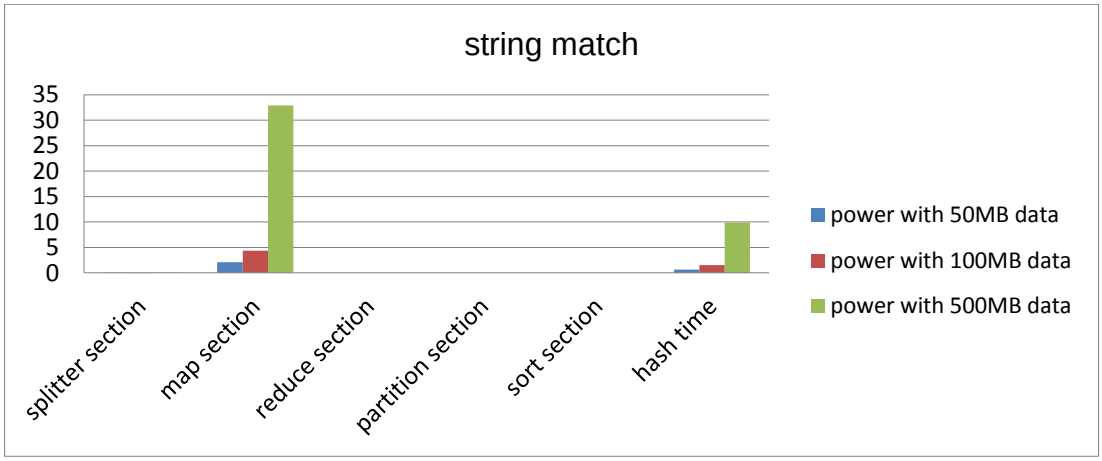


Fig .6 power consumption in string match application

First, data is divided into 3 categories: small, medium and large. And consumed power of each category for all programs under study is measured and recorded. Tables 2 to 4 show these measurements. To see the consumed power better, two more common and more important programs named "word count" and "string match" which cover all parts of the work are shown separately.

It is obvious from the above graphs and tables that most of the activities occurred in the map part and consumed power in map part of different programs is far from other phases of each program. In "word count" program most of the consumed energy is allocated to map. In sort part by changing the size of the data, by doubling the data, consumed energy increases more than two times, unexpectedly. So, it is expected that by increasing size of input data, consumed energy will paramount. In map part by doubling the size of the data, consumed energy also is doubled approximately.

In all programs, by increase in size of the input data, consumed power will be much more evident. In matrix multiplication program, by increasing the size of the data, calculation increased and the consumed power is fixed and by doubling size of the input, consumed power increased by less than 2 times.

In "string match" program, consumed of hash part is considerable and includes about ¼ of the total consumed energy. By increasing size of the input file, the consumed power increase non-linearly.

In programs which contain sort and hash, the rule of consumed power is more evident and by increasing input data, input of these tasks becomes more complicated. consequently, sub-programs of these operations should be

improved and better algorithms should be used in these operations.
In all programs consumed power of reduce part is not considerable. But it should be noted that most of the program either haven't reduce part or a little work is done in this part.

4. Conclusion

Today, consumed energy plays an important role in the world we are living in and we look for the cases which consume less energy. This role is of the same importance in computers and its programs. Moreover, volume of the data which should be processed is increasing such that we face explosion of the data.
Map reduce programming framework is used for processing these huge data. Different implementations of this framework in different environments are proposed.
One of these implementations is in multi-core environments with common memory. Here, measurements are done in this environment.
Based on the measurement following results obtained:
Most of the activities occurred in Map part and consumed power in Map part of different programs is much more than that of other phases of each program. In programs which include sort and hash parts, role of consumed power is more evident and by increasing input data, these tasks become more complicated. So, sub-programs of these operations should be improved and better algorithms should be used for these operations.

5. Future works

Measurement in Mars framework and comparison with Phoenix will be done in the future. And we will try to look for algorithms which improve the sort and hash functions and results of implementation of these algorithms will be investigated.

6. References

[1] J. Lo, J. Emer, H. Levy, R. Stamm, D. Tullsen, and S. Eggers. Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems*, 15(3):322–353, August 1997.

[2] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*. 2009

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation , San Francisco, CA, 2004.

[4] Soyean.park, weihang.jiang, yuanyuan.zhou, sarita. Managing Energy –Performance Tradeoffs for Multithreaded Applications ON Multiprocessor Architectures adve*SIGMETRICS'07* June 12–16, 2007, San Diego, California, USA.

[5] G. E. Moore, "Cramming more components onto integrated circuits", Proc of Electronics, vol. 38, pp. 114-117, April 1965.

[6] W. A. Wulf, S. A. McKee, "Hitting the memory wall: implications of the obvious", ACM SIGARCH Computer Architecture News,vol. 23, pp.20-24, March 1995
` | `
[7] Sh. Ryoo, Ch. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk and W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA", Proc. ACM SIGPLAN Symposium on Principles and practice of parallel programming, February 20-23, 2008, Salt Lake City, UT, USA

[8] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques (PACT) ,Toronto, Canada, October 2008.
[9] C. Belady. In the Data Center, Power and Cooling Costs More than the IT Equipment it Supports. *Electronics Cooling*, 23(1), 2007.

[10] Report To Congress on Server and Data Center Energy Efficiency. In *U.S. EPATechnical Report*, 2007.

[11] L. A. Barroso and U. H¨olzle. The Case for Energy-Proportional Computing.*IEEE Computer*, 40(12), 2007.
[12] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In Proceedings of the 13th Intl. Symposium on High-Performance Computer architecture (HPCA) Phoenix,AZ, February 2007.
[13] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques (PACT) ,Toronto, Canada, October 2008.

[14]willis.lang , jignesh m.patel ,Energy Management for Map reduce Clusters *Proceedings of the VLDB Endowment*, Vol. 3, No. 1Copyright 2010 VLDB Endowment

[15] K. Heafield, "Introduction To Hadoop," Google Inc, 2008

[16]y.chen,a.ganapathi,a.fox,r.katz,david.patterson,Statistical Workload for Energy Efficient Mapreduce, Technical Report No.UCB/EECS-2010-6,January21, 2010(http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-6.html)

[17] Apache Software Foundation. JIRA issue MAPREDUCE-776.Gridmix:Trace-basedbenchmarkforMap/Reduce. https://issues.apache.org/jira/browse/MAPREDUCE-776.

[18] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. In *HotPower '09: Workshop on Power Aware Computing and Systems*, 2009.

[19] Google. Data center efficiency measurements. The Google Blog, 2009.

Evaluation of Encryption Algorithms for Privacy Preserving Association Rules Mining on Distributed Horizontal Database

Ashraf El-Sisi, and Hamdy M. Mousa
Faculty of Computers and Information, Menofyia University, Egypt
{ashrafelsisi@hotmail.com, hamdimmm@hotmail.com }

Abstract - *Encryption algorithms used in privacy preserving protocols can be affected on overall performance. In this paper we study several encryption algorithms with two methods of privacy preserving association rule mining on distributed horizontal database (PPARM4, and PPARM3). The first method PPARM4 computes association rules that hold globally while limiting the information shared about each site in order to increase the efficiency. The second method PPARM3 is a modification for PPARM4 based on a semi-honest model with negligible collision probability. Common encryption algorithms for the two methods of privacy preserving association rule mining on distributed horizontal database selected based on performance metric. So a performance comparison among five of the most common encryption algorithms: RSA, DES, 3DES, AES and Blowfish with the two privacy methods are presented. The comparison has been conducted by running several encryption settings with the two methods of privacy preserving association rule mining on distributed horizontal database. Simulation has been conducted using Java. Results show that, PPARM3 gives better performance with all encryption algorithms implemented. Also PPARM3 with encryption algorithm DES gives best result with different database sizes. Based on the results we can tune the suitable encryption algorithm from our implementations to the required overall performance.*

Keywords: Encryption, distributed data mining, Association rule mining, privacy, security.

1 Introduction

Data Mining (DM) techniques have been widely used in many areas especially for strategic decision-making [1-8]. Apart from its usual benefits, it also has a few disadvantages associated with it. Experts say that data mining in the wrong hands will end up in destruction. The main threat of data mining is to privacy and security of data residing in large data stores [9-15]. Some of the information considered as private and secret can be brought out with advanced data mining tools. It is a real concern of people working in the field of database technology. Different research efforts are under way to address this problem of preserving security and privacy. The privacy term is overloaded, and can, in general, assume a wide range of different meanings. For example, in the context of the Health Insurance Portability and Accountability Act (HIPAA) Privacy Rule, privacy means the individual's ability to control who has the access to personal health care information. From the organizations point of view, privacy involves the definition of policies stating which information is collected, how it is used, and how customers are informed and involved in this process. We can considering privacy as "The right of an entity to be secure from unauthorized disclosure of sensible information that are contained in an electronic repository or that can be derived as aggregate and complex information from data stored in an electronic repository". There are many methods for privacy preserving distributed association rule mining across private databases. So these methods try to compute the answer to the mining without revealing any additional information about user privacy. An application that needs privacy preserving distributed association rule mining across private databases, like medical research. Sensitive information contained in a database can be extracted with the help of non-sensitive information. This is called the inference problem. Different concepts have been proposed to handle the inference problem. The process of modifying the transactional database to hide some sensitive information is called sanitization. By sanitizing the original transactional database, the sensitive information can be hidden. In the sanitization process, selective transactions are retrieved and modified before handing over the database to a third party. Modification of transaction involves removing an item from a transaction or adding an element to the transaction. In some cases, transactions will be either added to or removed from the database as suggested in [16]. The modified database is called sanitized database or released database. The efficiency of a privacy-preserving algorithm is measured based on: (1) the time taken to hide the data, (2) the number of new rules introduced because of the hiding process, and (3) the number of legitimate rules lost or unable to be extracted from the released database. Encryption algorithm used in privacy preserving can be affected on overall performance, so in this paper we addresses the problem of evaluate several encryption algorithms with two protocols of privacy preserving association rule mining (PPARM4, and PPARM3) on distributed horizontal database. The numbers (4 and 3) in

abbreviations (PPARM4, and PPARM3) respectively means the number of computation steps to get the results of protocol. The first protocol (PPARM4) computes association rules that hold globally while limiting the information shared about each site in order to increase the efficiency [17]. The second protocol (PPARM3) is a modification for the first based on a semi-honest model with negligible collision probability [18]. Section 2 gives an overview about the problem and the related work in the area of privacy preserving association rule mining on distributed homogenous databases. In section 3 some details of the two protocols of the algorithms of computing the distributed association rule mining (PPARM4, and PPARM3) to preserve the privacy of users. Sections 4 and 5 describe implementation and results of several encryption algorithms with the two methods of privacy preserving association rule mining on distributed horizontal database. Finally, some conclusions are put forward in Section 6.

2 Association Rule Mining

Association rule mining finds interesting associations and/or correlation relationships among large sets of data items. Association rules show attributes value conditions that occur frequently together in a given dataset. In [19] the association rules mining problem can formally be defined as follows: Let I = {i1, i2, . . . , in} be a set of items. Let DB be a set of transactions, where each transaction T is an itemset such that . Given an itemset , a transaction T contains A if and only if . An association rule is an implication of the form where and . The rule has support S in the transaction database DB if S% of transactions in DB contains. The association rule holds in the transaction database DB with confidence C if C% of transactions in DB that contain A also contains B. An itemset X with k items is called a k-itemset.

2.1 Distributed Association Rule Mining Problem

The problem of mining association rules is to find all rules whose support and confidence are higher than certain user specified minimum support and confidence. Clearly, computing association rules without disclosing individual transactions is straightforward. We can compute the global support and confidence of an association rule knowing only the local supports of AB and ABC, and the size of each database:

$$Support_{AB \Rightarrow C} = \frac{\sum_{i=1}^{numberofsites} Support\_count_{ABC}(i)}{\sum_{i=1}^{numberofsites} database\_Size(i)}$$

$$Support_{AB} = \frac{\sum_{i=1}^{numberofsites} Support\_count_{AB}(i)}{\sum_{i=1}^{numberofsites} database\_Size(i)}$$

$$Confidence_{AB \Rightarrow C} = \frac{Support_{AB \Rightarrow C}}{Support_{AB}}$$

Note that this requires no sharing of any individual transactions. And Protects individual data privacy, but it does require that each site disclose what rules it supports, and how much it supports each potential global rule. What if this information is sensitive? Clearly, such an approach will be secure under secure muti-party computation (SMC) definitions by some modification, a way to convert the above simple distributed method to a secure method in SMC model is to use secure summation and comparison methods to check whether threshold are satisfied for every potential itemset [18]. For example, for every possible candidate 1-itemset, we can use the secure summation and comparison protocol to check whether the threshold is satisfied. Fig. (1) gives an example of testing if itemset ABC is globally supported, it shows determining if itemset support exceeds 5 percent threshold. Each site first computes its local support for ABC, or specifically the number of itemsets by which its support exceeds the minimum support threshold (which may be negative).

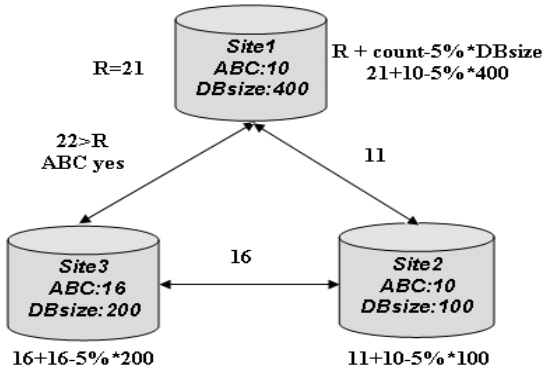


Fig. 1. Computing global support securely

The parties then use the secure summation algorithm (the first site adds a random (R) to its local excess support, then passes it to the next site to add its excess support, etc. and finally when pass to first site subtract the generated random from the result). The only change is the final step, the last site performs a secure comparison with the first site to see if the sum ≥ R. In the example, R -10 is passed to the second site, which adds its excess support (5) and passes it to site 3. Site 3 adds its excess support; the resulting value (22) is tested using secure comparison to see if it exceeds the Random

value (21). It is, so itemsets ABC is supported globally. Due to huge number of potential candidate itemsets, we need to have a more efficient method. This can be done by observing this lemma, (If a rule has support > k% globally, it must have support > k% on at least one of the individual sites). A distributed algorithm for this would work as follows, request that all rules are sent by each site with support at least k, for each rule returned, request that all sites send the count for their transactions that support the rule, and the total count of all transactions at the site. From this, we can compute the global support of each rule, and be certain that all rules with support at least k have been found. This has been shown to be an effective pruning technique. In order to use the above lemma, we need to compute the union of locally large sets. Then use the secure summation and comparison only on the candidate itemsets contained in the union. Revealing candidate itemsets means that the algorithm is no longer fully secure: itemsets that are large at one site, but not globally large, would not be disclosed by a fully secure algorithm. However, by computing the union securely, we prevent disclosure of which site, or even how many sites, support a particular itemset. This release of innocuous information (included in the final result) enables a completely secure algorithm that approaches the efficiency of insecure distributed association rule mining algorithms. The function now being computed reveals more information than the original association rule mining function. However, the key is that we have provable limits on what is disclosed.

3. (PPARM4, and PPARM3) Protocols Details

In secure multi-party computation approaches, given two parties with inputs x and y respectively, the goal of secure multi-party computation is to compute a function f(x, y) such that the two parties learn only f(x, y), and nothing else. In [19] there are various approaches to this problem. In [20] an efficient protocol for Yao's millionaires' problem showed that any multi-party computation can be solved by building a combinatorial circuit, and simulating that circuit. A variant of Yao's protocol is presented in [21] where the oblivious transfers is used to make secure decision tree learning using ID3 with efficient cryptographic protocol and their also two solution of our problem under the secure multi party computation for association rule mining [22], [23]. In [23] an explanation of a much more efficient method for this problem is described. To obtain an efficient solution without revealing what each site supports, they instead exchange locally large itemsets in a way that obscures the source of each itemset. They assume a secure commutative encryption algorithm with negligible collision probability. Intuitively, under commutative encryption, the order of encryption does not matter. If a plaintext message is encrypted by two different keys in a different order, it will be mapped to the same cipher text. Formally, commutatively ensures that Ek1(Ek2(x)) = Ek2(Ek1(x)). The main idea is that each site encrypts the locally supported itemsets, along with enough "fake" itemsets to hide the actual number supported. Each site then encrypts the itemsets from other sites. An example illustrate the protocol in [19] is given in fig. (2). Using commutative encryption, each party encrypts its own frequent itemsets (e.g., Site 1 encrypts itemset ABC). The encrypted itemsets are then passed to other parties, until all parties have encrypted all itemsets. These are passed to a common party to eliminate duplicates, and to begin decryption. (In fig. (2), the full set of itemsets is shown to the left of Site 1, after Site 1 decrypts). Then this set is passed to each party, and each party decrypts each itemset. The final result is the common itemsets (ABC and ABD in fig. (2)), an approach to proof that protocol preserves privacy can be found in [20]. This approach to prove that algorithm reveals only the union of locally large itemsets and a clearly bounded set of innocuous information.

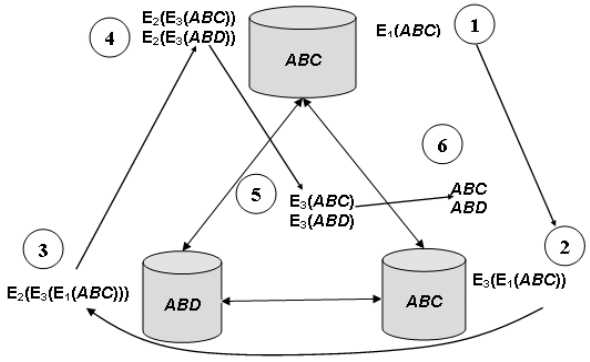


Fig. 2. Steps needed for computing the algorithm in [25]

3.1 PPARM4 protocol

Other method for privacy preserving association rule mining on distributed homogenous databases (PPARM4) in [17], showed that the protocol in [23] employs commutative encryption algorithm so it adds large overhead to the mining process then another protocol improves this by applying a public-key cryptosystem algorithm on horizontally partitioned data among three or more parties. In this protocol, the parties can share the union of their data without the need for an outside trusted party. Each party works locally finding all local frequent itemsets of all sizes. Then use public key cryptography to find the union of a frequent local itemset. We find that this method reduce the number of steps from 6 to 4 to calculate the global candidate item sets as shown in fig. (3) where K1 is private key and k2 public key . In [17] the results showed that this improvement reduces the time of mining process compared to method in [23]. For description this protocol, let P = {P0, . . . , Pn} be a set of N parties where |N| ≥ 3. Each party Pi has a database DBi. With assuming that parties running the protocol are semi-honest. The goal is to share the union of DBi as one shuffled database and hide the link between records in DBComp and their owners. This

employs a public-key cryptosystem algorithm on horizontally partitioned data among three or more parties. In this protocol, the parties can share the union of their data without the need for an outside trusted party. The information that is hidden is what data records where in the possession of which party. Protocol is described for one party as the protocol driver as shown in table [1]. The first party called Alice.

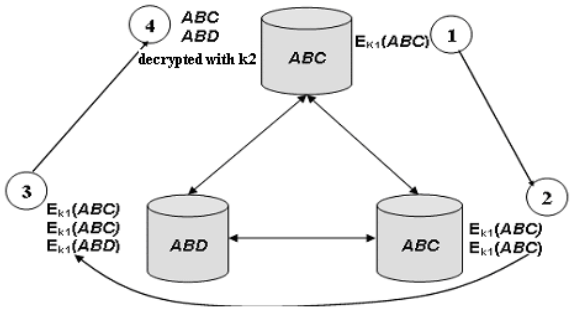


Fig. 3. Steps needed for computing PPARM4 algorithm

Table [1] Description of PPARM4 protocol

1. *Alice generates a public encryption key kPA.*
Alice makes kPA known to all parties (for illustration another two parties called Bob and Carol can be used).
2. *Each party (including Alice) encrypts its database DBi with Alice's public key.*
This means the encryption is applied to each row (record or transaction) of the database. Parties will need to know the common length of rows in the database. We denote the result of this encryption as kPA(DBi). Note that, by the properties of public cryptosystems, only Alice can decrypt these databases.
3. *Alice passes her encrypted transactions kPA(DB1) to Bob*. Bob cannot learn Alice's transactions since he does not know the decryption key.
4. *Bob mixes his transactions with Alice's transactions.*
That is, he produces a random shuffle of kPA(DB1) and kPA(DB2) before passing all these shuffled transactions to Carol.
5. *Carol adds and shuffles her transactions kPA(DB3) to the transactions received from Bob.*
6. *The protocol continues in this way, each subsequent party receiving a database with the encrypted and shuffled transaction of all previous parties in the enumeration of the parties. The i-th party mixes randomly his encrypted transactions kPA(DBi) with the rest and passes the entries shuffled transaction to the (i + 1)-th party.*
7. *The last party passes the transactions back to Alice.*
8. *Alice decrypts the complete set of transaction with her secret decrypt key.*
She can identify her own transactions. However, Alice is unable to link transactions with their owners because transactions are shuffled.
9. *Alice publishes the transactions to all parties.*
If the number of parties is N, then N − 1 of the parties need to collude to associate data to their original owners (data suppliers).

3.2 PPARM3 protocol

In [18] fast privacy preserving association rule mining on distributed homogenous databases (PPARM3), reduces the number of steps from four steps to only three steps for any numbers of clients to calculate the global candidate itemsets. The details of PPARM3 protocol as in fig. (4) and table [2].Table [3] shows a comparison of the three algorithms in [17, 18, and 23].

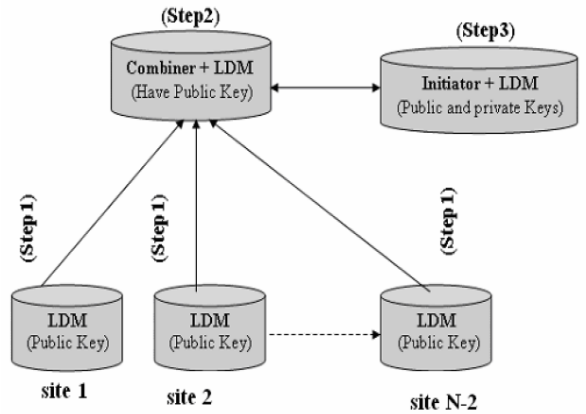


Fig. 4. General structure of PPARM3 algorithm

Table [2] Description of PPARM3 protocol

Protocol: Finding large itemsets of size k and global association rules.
Require: N >3 sites one site is algorithm initiator and another is data mining combiner and other called clients (local data mining) sites numbered (1..N − 2) and we assume negligible collision probability.
Step 1: *All local data mining (LDM) compute the mining results using fast distributed mining of association rules (FDM) [23] as locally large k-item sets (LLi(k)) and local support for each item set in LLi(k) then Encrypt frequent item sets and support (LLei(k)) then send it to the data mining combiner.*

Step 2: *The combiner merge all received frequent items and supports with the data mining combiner frequent items and support in encrypted form then send LLe(k) to algorithm initiator to compute the global association rules .*
Step3: *The algorithm initiator receives the frequent items with support encrypted. The initiator first decrypt it, then merges it with his local data mining result to obtain global mining results L(k), then compute global association rules and distribute it to all protocol parties.*

Table 3. Comparison between algorithms in [17, 18, and 23].

Comparison factors	Algorithm [23]	Algorithm [17]	Algorithm [18]
Steps for computation	6 steps	4 steps	3 steps
Rounds to compute results	2 rounds	2 rounds	1 round
Cryptography used	Communica tive	Public key	Public key
Mining algorithm	Apriori	Apriori	Apriori-Tid

3.3 Encryption Algorithms

Many encryption algorithms are widely available and used in information security. They can be categorized into Symmetric (private) and Asymmetric (public) keys encryption. In Symmetric keys encryption or secret key encryption, only one key is used to encrypt and decrypt data. The key should be distributed before transmission between entities. Keys play an important role. If weak key is used in algorithm then every one may decrypt the data. Strength of Symmetric key encryption depends on the size of key used. For the same algorithm, encryption using longer key is harder to break than the one done using smaller key [24]. There are many examples of strong and weak keys of cryptography algorithms like RC2, DES, 3DES, RC6, Blowfish, and AES. RC2 uses one 64-bit key. DES uses one 64-bits key. Triple DES (3DES) uses three 64- bits keys while AES uses various (128,192,256) bits keys. Blowfish uses various (32-448); default 128bits while RC6 is used various (128,192,256) bits keys [25-28]. Asymmetric key encryption or public key encryption is used to solve the problem of key distribution. In Asymmetric keys, two keys are used; private and public keys. Public key is used for encryption and private key is used for decryption (E.g. RSA and Digital Signatures). Because users tend to use two keys: public key, which is known to the public and private key which is known only to the user. There is no need for distributing them prior to transmission. Asymmetric encryption techniques are slower than Symmetric techniques, because they require more computational processing power.

4. Implementation of (PPARM4 and PPARM3) Methods

We implement PPARM4 and PPARM3 algorithms using java. Because the distributed association rules mining need the mining run in more than one site, we can use the RMI (remote method invocation) to connect the sites with each other. Our application is two parts one name server and other is client so we have two site works as server. First is the protocol initiator and second is the data mining combiner and we need client for every participant in the protocol. The initiator is responsible of the threshold of the mining algorithm so it need to define the support and confidence and also generate the public key (k2)and private key (k1) used in encryption and decryption in protocol and finally compute the final results. The data mining combiner responsible of combining the results of clients sites and mix the results to make better privacy of user data and every client is responsible of making the local data mining and encrypt the results of mining and send to data mining combiner. In our implementation our test in data that represent based on 0/1 matrix. And using Public-Key Cryptography, where each user places in a public file an encryption procedure. That is, the public file is a directory giving the encryption procedure of each user. The user keeps secret the details of his corresponding decryption procedure. There are several examples of commutative encryption; perhaps the most famous being RSA (if keys are not shared) and Pohlig-Hellman encryption. Firstly we use RSA that is useful to fulfill our requirements. After that we repeat the same work with different encryption algorithms (DES, 3DES, AES and Blowfish] to evaluate the best encryption algorithm improving the overall performance privacy preserving protocol.

5. Results of implementations and discussion

By running PPARM4 and PPARM3 algorithms 75 times. Running testing based on 5 homogenous data bases with different size from 2500 bytes to 2500000 bytes by 15 time for every data base. The 15 values are much closed to each other. The values listed in table [4] and tables [5] are the average values of time for PPARM4 and PPARM3 respectively in case RSA, DES, 3DES, AES and Blowfish encryption algorithms. Testing is performed using P4 (2.8 GHZ) with Java (SDK 1.6). Fig. (5) and fig. (6) shown the relation between different database sizes and time consuming of protocols (PPARM4 and PPARM3) for each encryption algorithm (RSA, DES, 3DES, ASE, and Blowfish) respectively. The results show PPARM3 is faster than PPARM4 in case all encryption algorithms implemented by the same ratio. Fig. (7) and fig. (8) show a comparison between time of protocol (PPARM4 and PPARM3) respectively and different encryption algorithms (RSA, DES, 3DES, AES and Blowfish) with variable database size. From these results we can say that the best performance of time for privacy protocols PPARM4 and PPARM3 is in case using encryption algorithm DES. So we can tune the required performance of privacy protocol by control in changing the encryption algorithm.

Table [4] Computation time in ms for PPARM4 with different encryption algorithms

Database size (B)	RSA	DES	3DES	AES	Blowfish
2500	0.11058	0.00055	0.00166	0.00083	0.00070
25000	1.21152	0.00606	0.01817	0.00909	0.00771
50000	2.35848	0.01179	0.03538	0.01769	0.01501
250000	11.6384	0.05819	0.17458	0.08729	0.07406
2500000	109.453	0.54727	1.64180	0.8209	0.69652

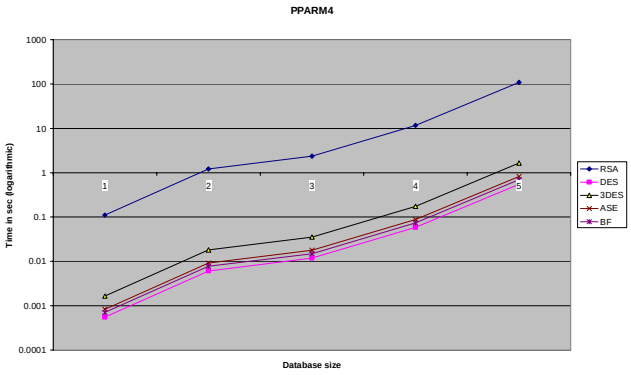


Fig. 5. Computation time in ms for PPARM4 with different encryption algorithms

Table [5] Computation time in ms for PPARM3 with different encryption algorithms

Database size (B)	RSA	DES	3DES	AES	Blowfish
2500	0.081684	0.00041	0.00123	0.00061	0.00052
25000	0.404043	0.00202	0.00606	0.00303	0.00257
50000	0.722407	0.00361	0.01085	0.00542	0.00460
250000	3.181355	0.01591	0.04772	0.02386	0.02025
2500000	53.23277	0.26616	0.79849	0.39925	0.33875

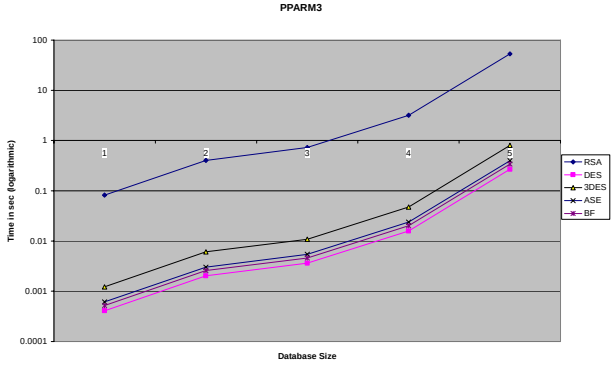


Fig. 6. Computation time in ms for PPARM3 with different encryption algorithms

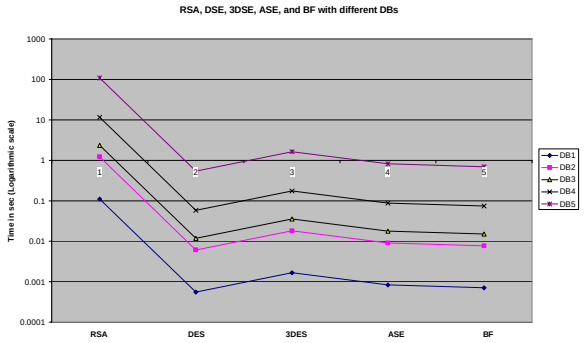


Fig. 7. Computation time in ms for PPARM4 with different database sizes

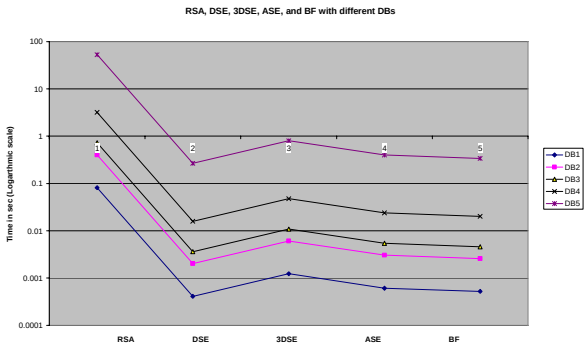


Fig. 8. Computation time in ms for PPARM3 with different database sizes

6. Conclusions

In this paper we presented study several encryption algorithms (RSA, DES, 3DES, AES and Blowfish) with different protocols of privacy preserving association rule mining on distributed horizontal database (PPARM4, and PPARM3). Testing is performed using P4 (2.8 GHZ) with Java (SDK 1.6). The results show PPARM3 is faster than PPARM4 in case all encryption algorithms implemented by the same ratio. The results prove that, the best performance of time for privacy protocols PPARM4 and PPARM3 is in case using encryption algorithm DES. So we can tune the required performance of privacy protocol by control in changing the encryption algorithm. In future work we can consider finer tuning by implement many different setting for encryption algorithm using with privacy protocol. For example longer key is harder to break than the one done using smaller key.3DES uses three 64-bits keys while AES uses various (128,192,256) bits keys. Blowfish uses various (32-448); default 128bits.

References

[1] Agarwal, R., Imielinski, T., & Swami, A. "Mining association rules between sets of items in large databases", In Proceedings of the ACM International Conferences on Management of Data pp. 207-216, 1993

[2] Agarwal, R. & Srikant, R., "Fast algorithm for mining association rules", In Proceedings of the 20th International Conference on Very Large Data Bases pp. 487-499, 1994.
[3] Fayyad, U. M., Piatetsky-Shapiro, G. Smyth, P., & Uthurusamy, R., "Advances in knowledge discovery and data mining", AAAI Press/The MIT Press. 1996.
[4] Han, J. W. & Kamber, M., "Data mining: Concepts and techniques", Morgan Kaufmann Publishers. 2001
[5] Han, J. W., Pei, J., & Yin, Y. W., "Mining frequent patterns without candidate generation", In Proceedings of the ACM International Conference on Management of Data pp. 1-12, 2000.
[6] Hidber, C., "Online association rules mining", In Proceedings of the ACM SIGMOD Conference Management of Data. 1999
[7] Lin, D., & Kedam, Z. M., "Pincer-search: An efficient algorithm for discovering the maximum frequent se", IEEE Transactions on Knowledge and Data Engineering, 14. 2002.
[8] Webb, G. I., "Efficient search for association rules", In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining pp. 99-107, 2000.
[9] Agrawal, R. & Srikant, R., "Privacy preserving data mining", In Proceedings of the ACM SIGMOD Conference, 2000.
[10] Agrawal, D., & Aggarwal, C. C., "On the design and quantification of privacy preserving data mining algorithms", In Proceedings of the ACM PODS Conference, 2001.
[11] Ashrafi, M. Z., Taniar, D., & Smith, K., "PDAM: Privacy-preserving distributed association-rule-mining algorithm", International Journal of Intelligent Information Technologies, (1), 49-69, 2005.
[12] Atallah, M., Bertino, E., Elmagarmid, A., Ibrahim, M., & Verykios, V., "Disclosure limitation of sensitive rules", In Proceedings of Knowledge and Data Exchange Workshop. 1999.
[13] Clifton, C., "Protecting against data mining through samples", In Proceedings
of the 13th IFIP WG11.3 Conference on Database Security, 1999.
[14] Lee, G., Chang, C., & Chen, A. L. P., " Hiding sensitive patterns in association
rules mining", In Proceedings of the 28th Annual International Conference on Computer software and Applications Conference pp. 424-429, 2004.
[15] Fatima M. and Safia N., "Privacy Preserving K-means Clustering: A Survey Research", International Arab Journal of Information Technology, Vol. 9, No. 2, 2012.
[16] Clifton, C. & Marks, D., "Security and privacy implications of data mining", In Proceedings of the ACM Workshop Data Mining and Knowledge Discovery, 1996.
[17] V. Estivill-Castro and A. Hajyasien "Fast Private Association Rule Mining by a Protocol Securely Sharing Distributed Data", Proceedings of the 2007 IEEE Intelligence and Security Informatics (ISI 2007), pp. 342-330, New Brunswick, New Jersey, USA, May 23-24, 2007.
[18] Ashraf El-Sisi, "Efficient Privacy Preserving Association Rules Mining Algorithm on Distributed Homogenous Data Base", International Arab Journal of Information Technology, Vol. 7, No. 2, 2010.
[19] D. Agrawal and C. C. Aggarwal, "On the design and quantification of privacy preserving data mining algorithms," in Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. Santa Barbara, California, USA: ACM, May 21-23 2001, pp. 247–255.
[20] O. Goldreich, "Secure multi-party computation," (working draft). [Online]. Available: http://www.wisdom.weizmann.ac.il/ oded/pp.html
[21] I. Ioannidis and A. Grama, "An efficient protocol for yao's millionaires' problem", Hawaii International Conference on System Sciences (HICSS-36), Waikoloa Village, Hawaii, Jan. 6-9 2003.
[22] Yehuda Lindell and Benny Pinkas, "Privacy Preserving Data Mining", Journal of Cryptography pp.177-206, 2002.
[23] M. Kantarcioglu and C. Clifton. "Privacy-preserving distributed mining of association rules on horizontally partitioned data", In IEEE Transactions on Knowledge and Data Engineering Journal, volume 16(9), pages 1026–1037, September 2004.
[24] Diaa S. Abdul. El., Hatem M. Abdul Kader and Mohie M. H., "Performance Evaluation of Symmetric Encryption Algorithms", IJCSNS International Journal of Computer Science and Network Security, VOL.8 No.12, pp. 280-286, 2008.
[25] W.Stallings, "Cryptography and Network Security", 4th Ed. Prentice Hall , PP. 58-309 . 2005.
[26] Coppersmith, D. "The Data Encryption Standard and Its Strength Against Attacks."I BM Journal of Research and Development, pp. 243-250, May 1994.
[27] Bruce Schneier, "The Blowfish Encryption Algorithm", Retrieved October 25, 2008, http://www.schneier.com/blowfish.html
[28] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", Communications of the ACM, vol. 21, no. 2, pp. 120–126, 1978.

Computer Network Reliability Dynamics Modeling: An Automatic Service Stabilization

Benson Moyo[1], **and Ndabezinhle Soganile**[2]
[1]Computer Science and Information Systems Department, University of Venda, Thohoyandou, Limpopo, South Africa; Email: benson.moyo@univen.ac.za
[2]Computer Science and Information Systems Department, University of Venda, Thohoyandou, Limpopo, South Africa; Email: ndabezinhle.soganile@univen.ac.za

Abstract - *Dependence on computer networks is now a reality. The networks are continuously becoming larger, heterogeneous and more complex in size and functionality. Any computer network service relies on several component interactions. This might involve mutually communicating hardware and software. The main task is to guarantee network service in the presence of network faults. In this work we present a design of a model to identify, predict, evaluate and neutralize faults on a computer network. The approach incorporates the capability of Dynamic Bayesian Networks to diagnose, predict and forecast faults and evaluate the magnitude of the network service degradation. The model is complex enough to diagnose and yet simple enough to avoid time and space complexity on the network. Fault thresholds to satisfy probability of error occurrence are established. We present model simulated results to demonstrate the applicability of proposed model.*

Keywords: Network reliability, fault, Dynamic Bayesian Network

1 Introduction

Dependence on computer networks has become a necessity to many organizations both profit making and nonprofit making including individuals. There is a whole explosion of services based on the network platform. The computer networks are themselves becoming larger, complex, ubiquitous, flexible and dynamic. Devices may be added, undergo repairs, be upgraded, and be removed from the network at any given time.

In order to access the correct service, a user needs several components to interact at least according to their requirement specification to perform one or more specified operational function. This might involve mutually communicating applications software, system software, protocols and communicating hardware and transmission media. The final output from the network is the sum total of the interaction of all the necessary components and subcomponents and protocols. Each component, each interface, each protocol plays an important role in the overall outcome. How can we guarantee high degree of network service survivability in the presence of network faults, malfunctions, malware attacks and software design flaws? The user is not interested in whether there has been a removal or repair of a device or debugging or upgrade of a protocol but is concerned with the availability of a an expected service at a given point in time as dictated by requirements and needs. Therefore organizations need some solid form of trust from the network. The confidence is brought by the application of sound reliability models.

In this work we present a design of a model to identify, predict, evaluate and neutralize faults on a computer network. In section 2 we present a brief overview of computer network faults classes. The Dynamic Bayesian Network approach to diagnose, predict and forecast faults and evaluate the magnitude of the network service degradation is also presented in section 2. We present model simulation results in section 3 and draw our conclusions in the subsequent section which is section 4.

2 Methodology

The body of work includes in depth analysis of the literature and work that has been done in the field of fault management. The relevant literature review guides the designing of computer network hybrid reliability model. Both secondary and primary sources of information are used to enhance the researchers' knowledge in the field of study and enable them to design a model based on the effort done by other researchers to the same end. Probabilistic techniques, Dynamic Bayesian Networks, and simulation methods are used as the foundation and the building blocks of this work. Network faults are identified using the Network Management error codes, and the Simple Network Management Protocol trap messages.

The simulation is designed and implemented using C++. The data used is collected from management agents via Simple Management Protocol messages (SNMP traps) and network system log-files. For simulation purposes the random occurrence of faults, the minimal standard random number generator as recommended in [1] is used. The probability outcomes are then compared with the statistics from the collected data as experimental control.

2.1 Overview of computer network faults

A network fault is defined as an abnormal condition or defect at the network component, equipment, or sub-system level which may lead to an error which may in turn lead to a failure [2]. Network faults constitute a class of network events that can cause other events but are not themselves caused by other events as explained in [2].

Therefore a network failure is a result of a network system state error and the generator of an error is a fault. An error can propagate through a network and cause hardware and software failure to otherwise faultless hardware and software subsystems. A failure is the manifestation of the error that is observable by the client. The client might be a human being or another network component or system [2].

Systems have faults but as long as that fault is not activated and it has not triggered an error, we cannot talk of a failure. Failure of a computer network is with respect to the abnormality in service provision visible to a client.

Computer network faults can be classified as permanent, transient and intermittent [2]. This is the taxonomy of faults based on their temporary effects. Taking an assumption that the active period of a fault is the interval during which the fault has a negative influence on the network, and benign period is the interval where the fault is not influential. Therefore a formalized classification definition is as follows: The permanent fault has an infinite active period. It reflects irreversible physical changes to a system or subsystem. Transient faults have a finite active time interval followed by an infinite benign period. These are generated by temporary network conditions like loss of signal in wireless network or an active attack by a hacker. Intermittent faults have finite active period and finite benign period, in other words these are recurring. Intermittent faults are internal in nature like network congestion.

Classification of faults based on the behavior are presented in [3], these can be summarized as crash faults (component either completely stops operating or never returns to a valid state); omission faults (component completely fails to perform its service); timing faults (component does not complete its service on time or suffers from synchronization) and Byzantine faults (faults of an arbitrary nature).

Although errors induced by transient and intermittent faults manifest very similarly, two main criteria as observed in [4] may be used to determine the type of the fault that generated the error. Firstly, failures generated by errors induced by intermittent faults tend to occur as a cluster at the same location, when the fault is activated. Secondly, replacement of the culprit component or subsystem removes the intermittent fault; by contrast failures generated by errors induced by transient faults cannot be eliminated by repair. The cause of a transient fault cannot be traced to a defect in a well identifiable part of the network. For simulation purposes we use some Simple Network Management Protocol Management Information Base (NMP MIB) data and network system logs to establish an estimate of transient fault rate and intermittent fault occurrence patterns. However this is highly contextual as each situation is dependent on a myriad of network characteristics from hardware brands and type to software and parameter configuration. A permanent fault induces a permanent error which in turn induces a permanent failure irrespective of the context. On the other hand a transient fault induces a transient error which subsequently induces a transient failure. Lastly an intermittent fault likewise induces an intermittent error which also triggers an intermittent failure.

According to Mouhammd [5], faults can be split into two categories, soft and hard faults. Wear or damage constitutes hard faults. These may be either hardware or software. The soft fault stem from poor engineering principles such as incomplete design. We focus on the soft faults in this study though there is overlapping.

Table 1: Fault classes

Faults Classes		
Permanent	Transient	Intermittent
Hardware clash Power outages Software clash	Malware Software bugs Hackers Buffer overflows	Packet loss Congestion End-to-end delay

The objective of understanding computer network faults is to be able to identify them according to their classes and as such it facilitates the modeling process.

Network reliability is the probability that a network will perform satisfactorily for at least a given period of time when used under stated conditions. It is an important attribute of a computer network as a system. Network reliability mostly deals with long term, or average behavior of the network. It is a property of the network and evaluates characteristics such as failure rate and failure density, architectural properties. The identification of faults is an important step towards reliability modeling of a system. In figure 1 below we show the relationship between failures, errors, and faults. A fault may develop into an error which may develop into a failure. However not all errors can develop into a failure. Some errors may only develop into failures only under certain conditions. The Millennium Bug, in Year 2000 is an example of a failure that was triggered by time.

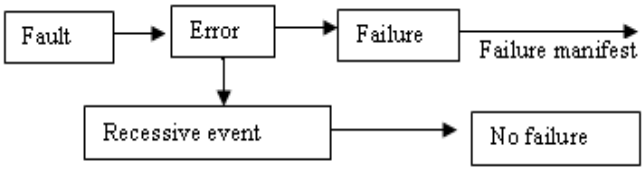


Figure 1: Fault, error failure relationship

A computer network is a repairable system. This means that if a component fails we can either replace the subcomponents that failed; this is referred to as repair maintenance, or replace the whole component by a new one

and this is referred to as preventive maintenance. All this is done to recover from a failure that would have been caused by a fault. In order to design a reliable system we need to specify the faults that the system may be subjected to. Therefore fault assumption is important in the whole system reliability modeling process. We have identified and classified the faults. Now we present methods for dealing with them.

2.2 Bayesian belief network

Bayesian networks provide a complete description of a given problem domain based on causal relationships. The cause-effect relationships enable both forward and backward reasoning. Every entry in the full joint probability distribution can be calculated from the information in the network. Bayesian networks represent full probability models in a compact and intuitive way. These networks can address problems in diagnosis, prediction, forecasting, information retrieval, knowledge representation and many other domains [6].

As well as being a complete and nonredundant representation of the domain, Bayesian networks are more compact than the full joint distribution and their time and space complexity is lower than that of other models like the Markov Chain Models. This property is what makes it feasible to handle domains with many variables. Bayesian networks are sparse systems [6]. This means that it satisfies the property of being locally structured. The premise for locally structured systems is that, each subcomponent interacts directly with only a bounded number of other components; regardless of the total number of components this will imply less complexity exponential explosion.

In the Bayesian Network framework the independence structure in a joint distribution characterized by a directed acyclic graph, with nodes representing random variables (which can be discrete or continuous, and may or may not be observable), and directed arcs representing causal or influential relationship between variables. The conditional independence assertions about the variables, represented by the lack of arcs, reduce significantly the complexity of inference and allow the underlying joint probability distribution to be decomposed as a product of local conditional probability distributions (CPD) associated with each node and its respective parents. The semantics of Bayesian networks can be viewed in two ways that is as networks that represent joint probability distribution or as an encoding of a collection of conditional independence statements. The two views serve to guide the construction of the network and the designing of inference procedures. [6].

2.2.1 The rationale behind employing Bayesian network for this study

As aforesaid Bayesian Networks, are space efficient data structures for encoding all of the information in the full joint probability distribution for the set of random variables that characterize a problem space. It uses the fact that in real-world problem domains, the dependencies between the variables are generally local. The Bayesian network allows one to compute any value in the full joint probability distribution of the set of random variables. It represents all of the direct causal relationships between variables which are an advantage in the determination of failure causes. It can be used to reason forward (top-down) from causes to effects (predictive reasoning) or backward bottom-up) from effects to causes (diagnostic reasoning). In other words we can infer faults sources from errors and vice versa.

2.2.2 Constructing Bayesian network for a network fault

Let X represents faults of arbitrary nature. Let the Bayesian Network for the set of fault variables $X=\{x_1, x_2, \ldots x_n\}$ represent a joint probability distribution: $P(x_1, x_2, \ldots x_n)$. Then we writing the joint probability as a conditional probability using the chain rule we have:

$$P(x_1, x_2, \ldots x_n) = P(x_n|x_{n-1}\ldots x_1)P(x_{n-1}\ldots x_1) \quad (1)$$

Then reducing equation 1 to each conjunctive probability to a conditional probability we have:

$$P(x_1, x_2, \ldots x_n) = \prod_{i=1}^{n} P(x_i | x_{i-1,\ldots.} x_1) \quad (2)$$

The expression is referred to as the chain rule. Therefore for every fault variable X_i in the network provided that Fault-Evidence $(X_i) \subseteq \{X_{i-1}\ldots X_1\}$, we can express its probability as:

$$P(X_i|X_{i-1}\ldots X_1)=P(X_i|\text{Fault-Evidence}(X_i)) \quad (3)$$

Let X_t, denotes a set of unobserved fault state in time t, and e_t denotes the observed error. The observed error implies fault evidence. Therefore to predict the future fault state of X we have to extend equation 3 to obtain the following expression:

$$P(X_{t+k}|e_{1:t}) \text{ for some } k > 0. \quad (4)$$

For example, the expression might mean computing the probability of network buffer overflow fault five time units from now given all the observations of network buffer overflow errors or failures up to now.

It is not only forward reasoning that is of interest but also retrospective reasoning. The diagnostic reasoning in helps establish fault sources. This fault hindsight entails computing the posterior distribution over a past fault state, given all errors or failures to the present point in time. That is, from equation 4, we wish to compute $P(X_k|e_{1:t})$ for some k time units such that $0 \leq k < t$. For instance if we want to compute the probability that there was network congestion five units of time ago given the network congestion based errors or failures up to now. In this case k=5 units of time.

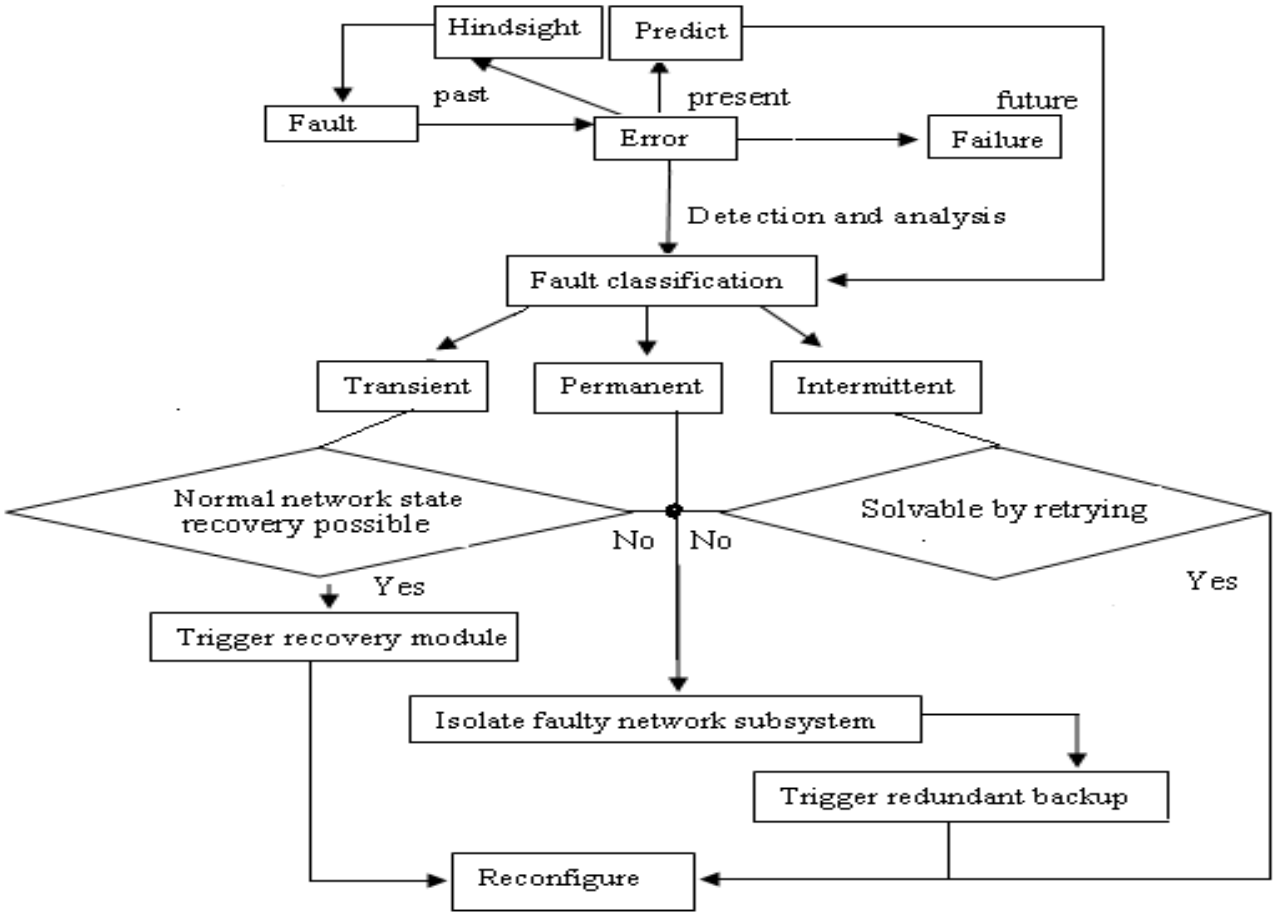


Figure 2: Proposed Reliability architecture

3 Results and discussion

Using the information obtained from a single organizations' network over a period of six months from SNMP, and network log files we obtained the fault rates shown in table 2. If there is a fault the probability that it might be associated with a protocol either poorly configured or corrupted is 0.48.

Table 2: Basic component fault rates from data collected

Basic network component	Fault rate
Switch	0.100
Bridge	0.130
Server	0.260
Network adapter	0.004
Router	0.210
Protocol	0.480
Transmission media	0.340

Let IF_{t-1} represent intermittent fault probability state variable in time t-1, and we also assume that: $P(IF_t|IF_{t-1}=true)=0.013$, $P(IF_t|IF_{t-1}=false)= 0.987$ and $P(IE_t|TF_t=true)=0.46$, $P(IE_t|IF_t=false)= 0.29$. IE_t represents the intermittent error in the network. We predict for the next nine time units (correct to 9 decimal places) and the results are shown in table 3 and figure 3. The results demonstrate the temporary nature of this type of fault. In some cases only a single run will result in the belief that a fault does not exist as proved by our simulation model. However due to its recurrence nature also we found that it can approach absolute values in some probabilistic instances.

Table 3: Intermittent fault probability over time

Time units into the future	Intermittent fault probability changes with time
1	0.110400000
2	0.006791759
3	0.001101247
4	0.000785880
5	0.000768500
6	0.000767543
7	0.000767490
8	0.000767487
9	0.000767487

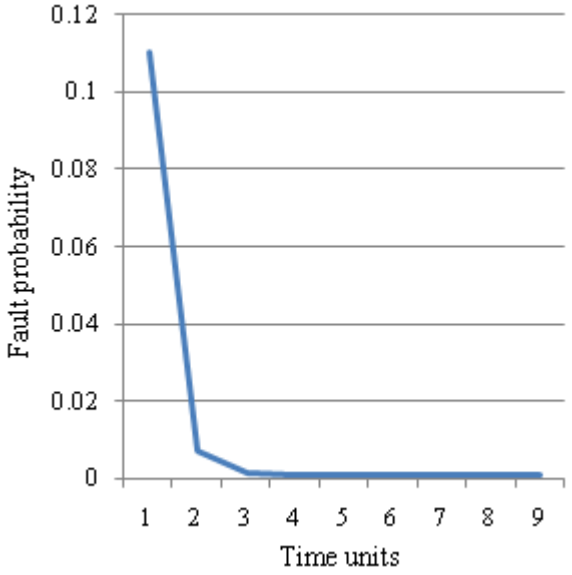


Figure 3: Intermittent fault probability trajectory over time

Given the initial belief that the probability of a transient fault is 0.5, TF_{t-1} represent transient fault probability state variable in a time t-1, and we also assume that $P(TF_t|TF_{t-1} = true) = 0.85$, $P(TF_t|TF_{t-1} = false) = 0.15$ and $P(TE_t|TF_t = true) = 0.49$, $P(TE_t|TF_t = false) = 0.027$. TE_t represents the transient error in the network. We predict for the next ten time units and the results are shown in figure 4. These results demonstrate the persistence of this type of fault and indicate a probability increase on the significant disruption of a particular network service. Figure 4 below shows the graphical representation of the results, this type of fault needs to be eliminated to restore normal service.

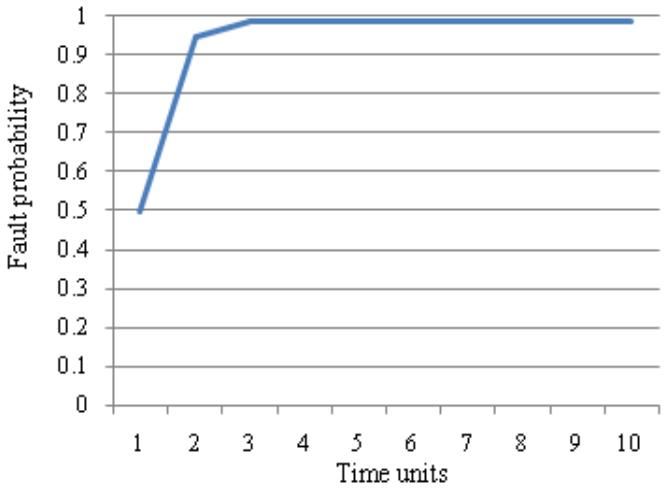


Figure 4: Transient fault probability progression over time

Table 4 was generated by our model simulation. However there is varying degrees of uncertainty when it comes to calculating the probability of a fault and classifying it as a condition necessary and sufficient for a failure occurrence.

Table 4: Forward inference on fault classification

Fault	Error	Failure type	Automated decision
Router interface out of synchronization	Packet loss	permanent	Temporarily isolate router by switching to standby router.
Timing mismatch	Time out	Intermittent, permanent or transient	Retry, or activate standby
Power cut	Blackout	Permanent	Switch to UPS
Broadcasting burst packets	Congestion , Bandwidth saturation	Intermittent	Retry
Network Protocol misconfigurations	Invalid routes	Transient	Switch to alternative node and reconfigure

4 Conclusions

In this paper we have considered the applicability of Bayesian networks for reliability modeling. Faults, errors and failures are probabilistic in nature therefore Bayesian Networks constitute a sound probabilistic rigorous mathematical modeling framework to tackle this domain. Network reliability is a research domain that will continue attracting research efforts as networks continue to grow in size functionality and complexity. Each component of a system can be engineered taking care of reliability; however when the systems are put together as subsystems to interoperate, challenges may immerge. Network configurations and reconfigurations, attacks, upgrades all introduce some level of network service uncertainty which is worthy researching. We have surveyed fault taxonomy in order to understand the characteristics and the dynamics of faults. The understanding will help architects to design networks that a more resilient to faults thereby guaranteeing service. The approach taken is based on the assumption that faults exists in any network what is important is how to handle them. We define network faults as a class of network events that can cause other events but are themselves not caused by other events.

The proposed model is designed to identify, localize, categorize the faults and select as well as initiate a recovery process. Prediction is achieved in order to preplan for the future abnormal behavior of the network. A level of redundancy by replicating some critical devices and network links to facilitate network restoration after localization and isolation of a component affected by a permanent fault is proposed.

In furthering this work it is recommended to research on fault parameter estimation and failure distributions and localization without depending on alarms from network management systems.

5 References

[1] J. Banks, B. L. Nelson, and D. M. Nicol, "Discrete-Event Simulation", Pearson Publishers, New Jersey, 5th Edition, 2010.

[2] M. Steinder, and A.. S. Sethi., "A Survey of fault localization techniques in computer networks", Journal of Science of Computer Programming , Vol No. 53, pp. 165-194, 2004, [online] Available from: http://citeseerx.ist.psu.edu/viewdoc, [accessed 20 February 2010]

[3] B. Selic, "Fault tolerance techniques for distributed systems" IBM, Software Group, 2004 [online], Available from: http://www.ibm.com/developerworks, [accessed 24 February 2009]

[4] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods", ACM Computing. Survey Vol. No. 42, Issue No. 3, Article 10 (March 2010), [online], Available from: http://delivery.acm.org/10.1145/1680000/1670680/a10-salfner.pdf , accessed 23 May 2010

[5] Mouhammd Al-Kasassbeh, and Mo Adda, "Analysis of mobile agents in network fault management", Journal of Network and Computer Applications Vol. No. 31, Issue No. 4. pp. 699-711, 2008

[6] S. J. Russell. and P. Norvig, "Artificial Intelligence: A Modern Approach", 2rd Edition, Prentice Hall , New Jersey, 2010

[7] H. Boudali, and J. B. Dugan, "A discrete-time Bayesian network reliability modeling and analysis framework", Journal of Reliability Engineering and Safety, Vol. No. 87, pp. 337-349, 2005 [online], Available from: http://www.sciencedirect.com, [accessed 12 July 2009]

[8] X. Jia, J. Cao, and W. Jia, "A classification of multicast mechanisms: implementations and Applications", The Journal of Systems and Software, Vol. No. 45, pp. 99-112, 1999.

[9] H. Li, and J. S. Baras, "Intelligent Distributed Fault and Performance Management for Communication Networks", Tech. Rep. CSHCN PhD 2002-2, Center for Satellite and Hybrid Communication Networks, University of Maryland, 2002. [online], Available from: http://www.isr.umd.edu, [accessed 12 August 2009]

[10] R. Maxion, "A case study of Ethernet anomalies in a distributed computing environment", IEEE Transactions on Reliability, Vol. No. 39, Issue No 4, pp. 433-443, 1990, [online] , Available from: http://www.ieeexplore.ieee.org [accessed 10 September 2009]

[11] G.R. Weckman, L.R. Shell, and J.H. Marvel, "Modeling the reliability of repairable systems in the aviation industry", Journal of Computer and Industrial Engineering Vol. No. 40, pp 51-63, 2001, [online], Available from: http://www.portal.acm.org, [accessed 5 June 2009]

A Parallel Formulation for the Simulation of a Generic Branch Predictor

L. Curi-Quintal[1,2] and J. Cadenas[1]
[1]School of Systems Engineering, University of Reading, Reading RG6 6AY, UK
[2]FMAT - Universidad Autonoma de Yucatan, Merida, Mexico
l.f.curiquintal@pgr.reading.ac.uk, o.cadenas@reading.ac.uk

Abstract - *A parallel formulation for the simulation of a branch prediction algorithm is presented. This parallel formulation identifies independent tasks in the algorithm which can be executed concurrently. The parallel implementation is based on the multithreading model and two parallel programming platforms: POSIX threads and Cilk++. Improvement in execution performance by up to 7 times is observed for a generic 2-bit predictor in a 12-core multiprocessor system.*

Keywords: branch predictor simulator, parallel formulation, multi-threading, Cilk++.

1 Introduction

A branch predictor is an algorithm implemented in hardware which is intended to improve the performance of instruction execution in the pipeline of modern microprocessors. This algorithm predicts whether a branch instruction should be taken (T) or not-taken (NT) by the microprocessor, based on stored history of execution of previous branches [1]. Prediction accuracy is inherent to the prediction algorithm and is frequently evaluated and quantified by computer simulation, using branch traces as input data. These branch traces contain a set of branch addresses and outcomes which represent every branch instruction seen by a processor when executing a computing task.

Typically, branch traces are collected from the execution of well-known benchmark programs, and stored in very large files with millions of addresses and outcomes. The simulation process executes a software model of the prediction algorithm and relates obtained results with observed outcomes. The analysis computes a prediction to each branch in the trace, compares this prediction with the actual outcome observed, and keeps a global tally of the number of comparison matches, referred to as hits. Behaviour and accuracy analysis of a branch prediction algorithm implies the execution of a batch process for a number of trace files, each one subject to different parameter values. This analysis requires a significant computational time.

In this paper, a parallel formulation of a generic branch predictor algorithm is described. This parallel formulation exploits the inherent parallelism of the algorithm and reduces the execution time of the analysis by simulation of a branch predictor using multiprocessor systems. The implementation of the parallel formulation has three main steps. Firstly, a concurrent classification of the branch traces, according to the branch address, is performed. Secondly, the execution of the predictor algorithm on each class of address proceeds concurrently. Thirdly, a final step to tally global hits is computed. Results show an improvement in execution performance by up to 7 times when this implementation is evaluated on a 12-core system.

2 Generic Branch Prediction Algorithm

Prediction algorithms are basically defined by two processes: a prediction function and an update procedure. The prediction function makes a prediction (T or NT) for a branch instruction based on information stored from previous branches. The update procedure modifies the recorded history of branches based on the prediction and the actual outcome of the branch. A Branch Prediction Buffer (BPB) maintains information of previous branches as a table. BPB is indexed by a hash function of the branch instruction address. BPB typically contains a set of bits indicating whether the branch was recently taken or not [1].

Fig. 1 describes an algorithm of a generic branch predictor simulator, using a BPB table with 2^S entries, with $S > 0$. Each entry in BPB table stores history for a branch instruction address.

Input arrays *A* and *O* in Fig. 1 contain the branch traces for the simulation. Variable *hits* stores the tally on the correct predictions. Every branch instruction in the trace is related with one element of the BPB table and evaluated by

the prediction function (lines 2-3) making a Boolean decision as either taken (T) or not-taken (NT). This value is compared with the recorded outcome (line 4), and matches are stored in *hits* (line5). Finally, BPB table is updated (line 7) at the corresponding entry for the branch instruction address.

```
Input:
   A ← array of addresses
   O ← array of outcomes
Output:
   hits: correct predictions

1. for i=1 to N do
2.  entry ← A_i mod 2^S
3.  p ← prediction(entry,BPB)
4.  if (p=O_i) then
5.       hits ← hits + 1
6.    end if
7.    update(entry,p,O_i,BPB)
8. end for
```

Figure 1. Pseudocode of a generic branch predictor simulator algorithm.

2.1 Parallel formulation

The pattern access to BPB table determines the behaviour of the algorithm in Fig.1: each branch instruction accesses only the entry in BPB corresponding to its address. Therefore, data decomposition can be applied sequentially over branch instruction addresses to classify outcomes. This classification creates a list of arrays, where each item in the list corresponds to outcomes for a specific address value in BPB. Thus, each array in the list can be processed by an independent predictor task, and all these tasks can be executed concurrently, although every individual task performs sequentially.

An example of this data decomposition is shown in Fig. 2. Arrays *A* and *O* represent the input traces. Array *A* contains a set of 13 hash values of branch instruction addresses, for a BPB table with 8 entries (entry values between 0 and 7). Array *O* contains its corresponding set of outcomes (T or NT).

i	1	2	3	4	5	6	7	8	9	10	11	12	13
A	5	6	2	4	6	1	7	1	5	3	5	3	7
O	T	T	NT	NT	T	T	T	NT	NT	NT	T	T	NT

LO

0	1	2	3	4	5	6	7
	T	NT	NT	NT	T	T	T
	NT		T		NT	T	NT
					T		

Figure 2. Example of data decomposition of input traces.

The input data is arranged in a list of arrays, *LO*. Each item in list *LO* corresponds to an entry value in BTB table, and contains an array of outcomes ordered by appearance in the trace. For example, for an entry value of 5, *LO* item has three outcomes (T, NT, T) corresponding to the outcomes of elements 1, 9 and 11 in the input trace.

Fig. 3 describes the parallel formulation for the algorithm presented in Fig. 1. In this parallel formulation, outcomes of the branch traces in array *O* are classified in a list of 2^S arrays (*LO*) according to the value of its corresponding address (lines 1-4). Concurrent tasks process *LO* items. Each task *k* executes sequentially the predictor algorithm on the array elements (lines 5-13) and registers correct predictions in $lhits_k$. After all tasks conclude, all *lhits* are accumulated in a global tally (*hits*).

```
Input:
   A ← array of addresses
   O ← array of outcomes
Output:
   hits: correct predictions

1.  for i=1 to N do
2.  entry ← A_i mod 2^S
3.  LO_entry.append(O_i)
4.   end for
5.   for each k in[0..2^S-1] do parallel
6.  while (LO_k.get(O_ki)) do
7.     p ← prediction(k, BPB)
8.     if (p=O_ki) then
9.    lhits_k = lhits_k + 1
10.    endif
11.    update(k,p,O_ki,BPB)
12.  end while
13. end foreach
14. for k=0 to 2^S-1 do
15.  hits = hits + lhits_k
16. end for
```

Figure 3. Pseudocode of the parallel formulation for the simulator of a generic branch predictor.

3 Parallel implementation

The parallel formulation algorithm in Fig. 3 was implemented using a shared memory model for concurrent tasks in multiprocessor systems called multithreading [2]. Multithreading is a parallel programming model that allows concurrent execution of multiple threads in the same process. A thread is a sequence of instructions within a process that can be scheduled for independent execution with other threads. Every program has one main thread. This thread can perform all the tasks by itself, or create other threads with defined subtasks. These subtasks should be designed to encapsulate functionality in order to exploit the inherent concurrency in the algorithm. Thread synchronization operations influence on the overall execution time performance [3]. Two multithreading programming platforms were used to compare performance of parallel implementations: POSIX threads and Cilk++.

3.1 POSIX threads

POSIX threads (*pthreads*) API (Application Program Interface) is a standard approved by IEEE for thread management. *Pthreads* is a portable threading library designed to provide a consistent programming interface across different operating systems platforms. Their main functions focus on thread creation, destruction and synchronization. The most common functions for thread synchronization are the mutual exclusion (*mutex*) locks and barriers. [4]

The *pthreads* version for parallel simulation of the branch predictor algorithm used in this paper encapsulates the classification of the outcomes and the predictor procedure. In the classification step, input arrays are divided in sub-arrays and each thread classifies a sub-array in a local version of *LO* list. After this classification is completed, corresponding arrays from all local *LO* lists are orderly merged in global arrays to create a global *LO* list. Then, the predictor function is executed concurrently, where each thread sequentially processes one item of the global *LO* list at a time. Finally, every thread updates the global tally of correct predictions with its own local tally.

In this implementation, barrier synchronization is used at the end of the input classification and at the end of merging local *LO*s. *Mutex* synchronization is used for concurrent update of the tally of correct predictions.

3.2 Cilk++

Cilk++ is a language extension for programming languages C/C++. Three statements make up the main part of the extension; *cilk_spawn*, *cilk_sync* and *cilk_for*. *cilk_spawn* and *cilk_for* are used to create parallel tasks, either with complete functions or with loop iterations. The *cilk_sync* statement is a local barrier, and is used to synchronize parallel tasks created by *cilk_spawn*. Aditionally, Cilk++ includes a library for mutex locks. Locking tends to be used much less frequently than in other parallel environments, such as *pthreads*, because all protocols for control synchronization are handled by the Cilk++ runtime system. The Cilk++ runtime system is based on a work-stealing scheduler using threads. This is a dynamic load-balancing scheduler and improves the utilization rate of processing units in a system [5].

The Cilk++ version for parallel simulation of branch predictor algorithm used in this paper implements three stages: input array classification, merging of local *LO* and prediction with verification. Every stage is implemented with a *cilk_for* statement. *cilk_for* is a replacement for the conventional C++ *for* statement and executes loop iterations in parallel. Cilk++ compiler converts a *cilk_for* loop into a divide-and-conquer recursive function encapsulating the loop body. This strategy benefits Cilk++ scheduler performance [6]. After the three *cilk_for* loops, the sum of all the partial tally of correct predictions is performed sequentially.

4 Results

Both parallel implementations were executed on an Intel 12-core (dual 6-core Xeon X5690, 3.47GHz) desktop system with Scientific Linux 6.0 (release 2.6.32-131). *Pthreads* version was compiled with gcc 4.4.4 (Red Hat 4.4.4-13), and Cilk++ version was compiled with cilk++ (GCC) 4.2.4 (Cilk Arts build 8503). Branch traces were collected from SPEC 2000 benchmark programs, with sizes of 10 million and 30 million of traces. BPB table was set at 4096 entries. We specifically exercised a 2-bit branch predictor algorithm [1].

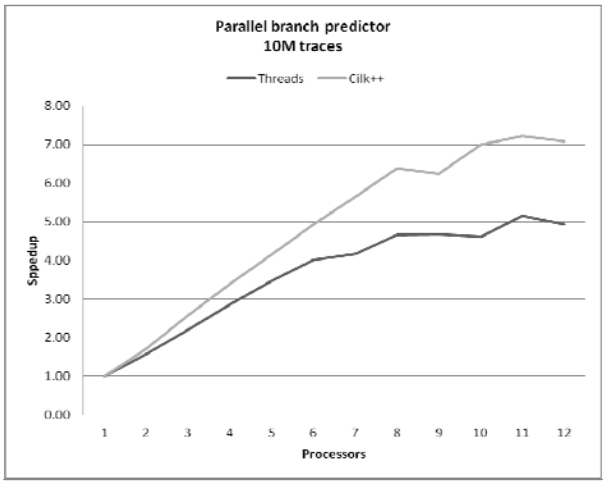


Figure 4. Speedups of the simulation of the parallel branch predictor in a 12-core system with 10 million traces.

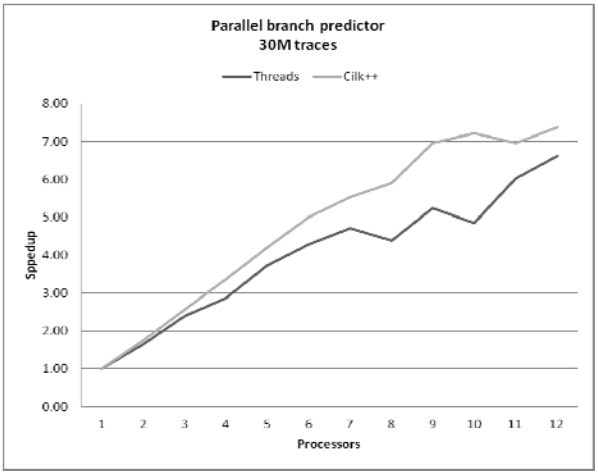


Figure 5. Speedups of the simulation of the parallel branch predictor in a 12-core system with 30 million traces.

Fig. 4 and Fig. 5 show the execution time speedup for *pthreads* and Cilk++ versions with input traces of 10 million and 30 million of elements, respectively.

On each parallel execution, one thread was created for each executing core. The execution time was calculated by the average of 50 executions of every version of the program (sequential, *pthreads* parallel and Cilk++ parallel).

5 Discussion

The Cilk++ implementation shows better performance than the *pthread* implementation for both sizes of branch traces files. Speedup in both implementations exhibits a sustained improvement on the first group of 6-cores. However, this improvement starts to decline with the second group of 6-cores. This behaviour is due to both the distribution of the data on cache memory of each processor and a slower external buffer that interconnects processors and memory.

The speedup tendency is very similar for Cilk++ implementation with both sizes of input traces. However, for *pthreads* implementation, with 10 million input traces, speedup starts to decline when the number of threads increases. With 30 million traces this tendency is not observed. This behaviour reflects how the cost of creating and synchronizing threads influences the execution performance.

The increasing speedup in performance suggests that concurrent independent tasks have been effectively identified from the sequential algorithm.

6 Conclusions

A parallel formulation for the simulation of a 2-bit branch prediction algorithm has been proposed. This parallel formulation was implemented, based on the multithreading model, using *pthreads* and Cilk++ programming platforms. Execution time performance exhibits an improvement with an incremental tendency by up to 7 times for a 12-core multiprocessor system. These results suggest that the parallel formulation effectively identifies inherent parallel tasks in the algorithm.

Future work is aimed at using the parallel formulation of the simulation of the 2-bit branch predictor with other branch predictor algorithms to analyze its performance.

7 Acknowledgements

This work was supported by PROMEP and UADY.

8 References

[1] Patterson, D.A., and J.L. Hennessy. *Computer Organization and Design: The Hardware /Software Interface, Fourth Edition*. Morgan Kaufmann Publishers, 2009.

[2] Rauber, T. and G. Runger. *Parallel Programming: for Multicore and Cluster Systems. First Edition*. Springer, 2010.

[3] Akhter, S. and J. Roberts. *Multi-Core Programming. Increasing Performance through Software Multi-threading*. Intel Press, 2006.

[4] Butenhof, D. *Programming with POSIX Threads.* Addison-Wesley Professional, 1997.

[5] Leiserson, C. E. *The Cilk++ concurrency platform.* Proceedings of the 46th Annual Design Automation Conference, 2009 (DAC '09), pp.522-527.

[6] Intel. *Intel Cilk++ SDK Programmer's Guide*. Document Num.: 322581-001US. Intel Corp, 2009.

Using SCPN for Modelling a Crossbar Switched Fabric CAN Network

Mohamed Mazouzi[1], **Ihsen Ben Mbarek**[2], **Oussama Kallel** [3], **Mohamed Abid**[4]
[1]Computer and Embedded Systems Laboratory, ENIS, Sfax, Tunisia
[2] Communication Systems Research Laboratory, ENIT, Tunis, Tunisia
[3] Communication Systems Research Laboratory, ENIT, Tunis, Tunisia
[4]Computer and Embedded Systems Laboratory, ENIS, Sfax, Tunisia

Abstract – *In recent years, the number of Electronic Control Units (ECU) steadily increases which require higher communication bandwidth. Switched fabric has become an active area of research because of its wide uses in industry. In fact, its uses can be a fast and reliable hardware solution for existing CAN-Bus problems like limited bandwidths and throughput.*
In this paper, we proposed and modeled a switched fabric CAN network Architecture based on CAN Controllers and switched fabric by the use of timed colored Petri nets (CPNTools).

Keywords*: Higher communication bandwidth, CAN Bus, Switched fabric, Switched fabric CAN Network, timed colored Petri nets, CPNTools.*

1 Current bussed Network problem and Switched Fabric CAN Network benefit

During last decades, the demand for sophisticated embedded systems requires the use of many connected equipments. A dedicated network bus [1] is used for connecting sensors, actuators in vehicles, robots and industries. Many serial buses were developed by car makers like MOST, J1850, SAE J1708, Byteflight, LIN... and CAN(controller Area Network). Most of them are specific to manufactures and not standardized.CAN is one of the most popular fieldbuses [2,3,4]. More than 400 million nodes were sold worldwide. It is used in those applications that require fast and reliable communication [5]. Nowadays, more sophisticated buses are concurrent to CAN networks like FlexRay [6], recently appeared, and RTethernet. They offer higher speed to satisfy the high bandwidth required for modern vehicles, suitable for x-by-wire application. In contrast the usage of FlexRay [7] is not widely used due to its complex specification and high cost.
Current parallel bus-based [8] solutions present some problems. In fact, it's well known that the physical separation of cards is limited to usually less than 3 feet. There are also limited bandwidths, high protocol overhead and no deterministic performance.

The limitations of a bussed network [2] are eliminated with crossbar switch network. A switched-fabric bus is unique in that it allows all CAN Controllers on a bus to logically interconnect with all CAN Controllers on the bus. The switching fabric is the physical connection within a switch between the input and output ports; it can be proved that all switches need a crossbar inside their switching fabric which allow them to operate at very high speed. Crossbar switches are widely used because of their simplicity and their high-performances [9] which promise to greatly simplify efforts and to add better capability and availability. Crossbar switch [8] can support simultaneously multiple messages. This greatly increases the aggregate bandwidth of the system. Because of the broadcast nature of the CAN protocol (ie: messages are not sent to a specific destination address, but rather as a broadcast), the chosen crossbar switch (as it is shown in Figure 1) is configured by closing all its crosspoints to ensure that the CAN message will be sent at the same time [3,4] for all outputs nodes as it is defined in CAN protocol [5].

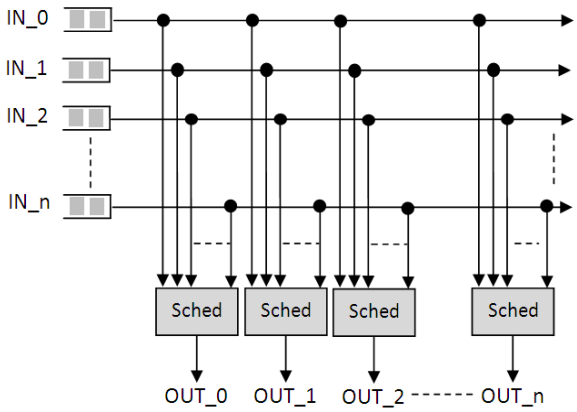


Fig. 1: NxN crossbar Switched Fabric CAN Network supporting broadcast

Each Electronic Control Unit (CAN Controller Node) produces a class priority of messages. For example, ECU_0 produces high level of priority and ECU_n (n in our model is equal 3) produce low level of priority. In fact, Produced CAN messages will be queued in the input queue of the incoming interface (If the input queue is full, the packet is dropped.). Therefore, to respect the CAN protocol philosophy, CAN messages will be broadcasted for all output port through crossbar Switched Fabric [10]. Furthermore, to reduce congested output port and to respect

the priority policy, each CAN message will be queued in the suitable output queue of each outgoing interface according to his level priority. (If the output queue is full, the packet is dropped). Then, each output port scheduler will select the message to be sent among the existing CAN message in accordance with his priority.
In our work, we modeled the switched fabric CAN Network using stochastic colored Petri. Our major contribution is to raise the lack of the bus solutions by proposing switched Fabric CAN. In fact, CAN based Networks using crossbar Switched fabric [11] have yet a well period before its replacement and it can compete the new sophisticated buses.

Our paper is organized as follow:

• The section 2 gives a short overview of Stochastic and Colored Petri Net SCPN [12].

• Based on the proposed architecture and CPNTools software, we model, in the last section, the most important Switched fabric CAN Network modules.

In the fourth section we give some conclusions of our work.

2 Short Recall of Coloured Petri Net

Coloured Petri Nets have been developed by K. Jensen in course of his PhD thesis (Jensen, 1980) to expand the modeling possibilities of classical Petri Nets. Like other forms of Petri Nets a CPN consists of places, tokens, transitions and arcs.
The primary feature unique to CPNs is the inclusion of evolved data structures into tokens [13,14]. These data structures are called coloursets and resemble data structures in high level programming languages; they can range from simple data types such as integers to complex structures like structs or unions in C/C++. Similar to programming languages it is possible to define variables associated with these coloursets such as linked list and queue.
Some examples of colourset and variable definitions are shown in Fig.2. Tokens as well as places of a CPN are always associated with a colourset and a place may only contain tokens of the same colourset as its own. To well understand the SCPN models of our Switched CAN controller, we give a short recall of CPN concepts.

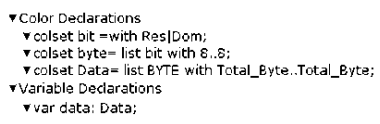


Fig. 2: Coloured and variable definitions

The places in a CPN are depicted as ellipses (Fig.3) with the name of the place written into it and the associated colourset (Id) below. A token in a CPN is represented by a small circle (Fig. 3). Its value (the data stored in the token) is shown in a rectangle attached to the circle. A number in the circle denotes the number of tokens with the same value. Figure 3 for example shows a place called *Buffer_Node_1* associated with the colourset CAN_Messages and holding one token with a value of{ {ID=[Dom,Res,Dom,Dom,Res,Dom,Dom, Dom,Res,Res,Dom,Dom,Dom,Res,Dom,Res,Dom,Dom,Do m,Res,Res,Dom,Res,Res,Res,Res,Res,Dom,Res],DATA=[b yte(4),byte(6),byte(5),byte(1),byte(5),byte(6),byte(6),byte(1)],TS=0}].

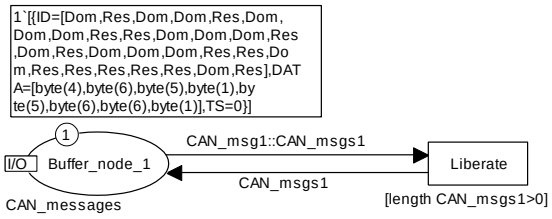


Fig. 3: Graphical representation of a place in CPN

Transitions in a CPN are represented by rectangles (Fig. 4) and can access the data stored in tokens by mapping tokens to variables. There are two possibilities to access this data:

-Guard conditions: The transition is enabled only if a specific condition – called a guard condition – regarding one or more variables is met. Guard conditions are encased in brackets and written above the transition (Fig.4).

-Transfer function: The transition reads and writes variables according to a specified function that can range from simple addition of values to complex conditional commands.

-Transfer functions consist of the definition of input () variables, output () variables and the commands to be carried out (action ()) and are attached below the transition (Fig.4).The example depicted in Figure 4 shows a transition that only fires if the length of variable *CAN_msgs* is less than the value *FIFO_length* and generates an output variable *CAN_msg* without taking any input variables (Fig. 4), the variable *CAN_msg* is filled with the return value of the function defined in the action part, *new_MSG_0*, which in this case is defined in the CPNtools area Declarations.

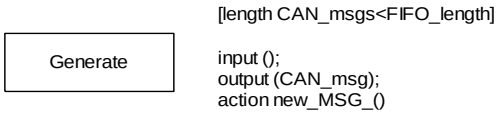


Fig. 4: Transition Generate with guard condition and transfer function

Places and transitions in a CPN are linked by arcs. Arcs in a CPN can be unidirectional or bidirectional. Unidirectional arcs transfer tokens from a place to a transition or vice versa.
Bidirectional arcs transfer the same token from a place to a transition and back. Arc inscriptions define the mapping of tokens to variables. An inscription can either be a constant value or a variable of the colourset that is associated to the place the arc is connected to. If all places connected to a transition by unidirectional input arcs or by bidirectional arcs hold tokens and its (optional) guard condition is met, the transition is said to be enabled. In case of more than one enabled transition in a CPN the one to fire is chosen randomly. Later on, we will add more places to our controller models to avoid arbitrary transitions.

For an analysis of clocked systems it is possible to define timed colourset, defined by the keyword timed and transition or arc delays marked by the characters @+.
If a colourset is defined as timed, a timestamp is added to the tokens of this colourset. The timestamp cannot be accessed by guard conditions or transfer functions. When

using timed colourset the firing of transitions depends on a global clock counter. Transitions can only fire if the clock value is the same as the largest timestamp of its input tokens. When a transition fires with a timed arc, the timestamp of its output token is the sum of the current clock value and the arc delay, in the example in Figure 5 this delay is *Time_sched_delay* clock cycles.

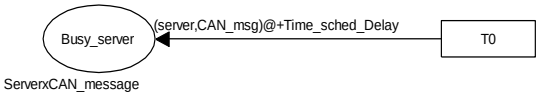


Fig. 5: Timed arc inscription.

3 SCPN Based Switched Fabric CAN Network model

In order to facilitate modeling of Switched CAN Controller, a modular approach was taken making use of hierarchical CPN models. The model of the Switched Controller is built following a hierarchical and modular architecture.

The root of the hierarchical representation of the model is shown in the figure 6.

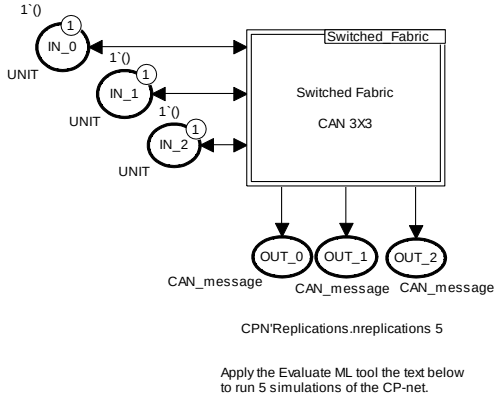


Fig. 6: Root level of the model

The Switched Fabric CAN 3x3 whose activity is modeled by the transition *Switched_Fabric* transmits the CAN message via the Switch Fabric. The places *IN_i* and *OUT_i* (i can be a value between 0:2) play the role of inputs/outputs for sub-models.

Nodes in CAN are identified by their identifier (in this model, colourset *Id* is a list of 29 bits). The coloursets and variables used in this model are shown in Figure 7.

Messages sent through the Switched Fabric CAN are represented by tokens of the colourset *CAN_message*. This colourset is a record of the colourset *Id* that designates the message priority and the colourset Data which represent the data field to be transmitted and the colourset *TS* for saving the time stamp for the birth of the message.

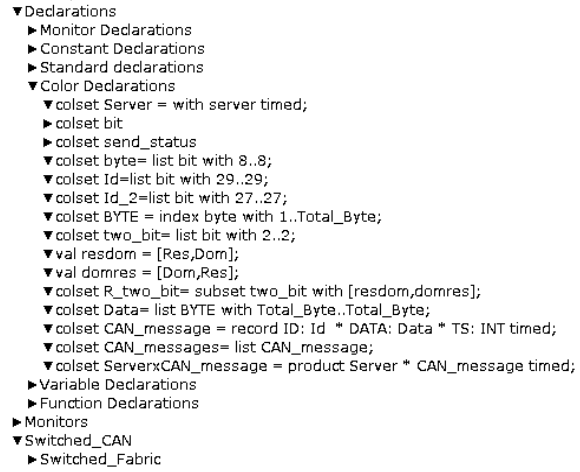


Fig. 7: Coloursets for CAN Network model

The variables (*CAN_msg*, *CAN_msg1* and *CAN_msg2*) are of type of the colorset *CAN_message*. This variable models the messages which cross the different sub-models of Figure 8 (*Node_i*, *Broadcast_i*, *FiFo_i_j* and *Scheduler_i*).

The Switched CAN network model in Figure 8 is composed of three nodes. Each node is represented by a transition and two places. The transition called *Node_i* (i can be a value between0:2) is a hierarchical transition which describes the messages generation within the node, how the messages are stored in buffer. The place *Buffer_Node_i* is used to store the messages already generated.

This place is configured with colourset *CAN_messages* which is a list of colourset *CAN_message*. When a token is present on this place (*Length CAN_msgs* >0) a message is ready for sending. This last fires the hierarchical transition *Broadcast_i*. The originated message is duplicated in three places, one for each output port of the Switch fabric. According to the priority which is associated to the messages (defined by their ID), the messages are stored in the FIFO queue (there is as many queue as of priorities). In this model, three levels of priority are defined: 0: high level of priority; 1: medium priority and 2: low priority.

FiFo_i_j is a substitution transition which presents the queue of *input_i* for the *output_j*. Finally, the scheduler processes the different messages according to its scheduling policy.

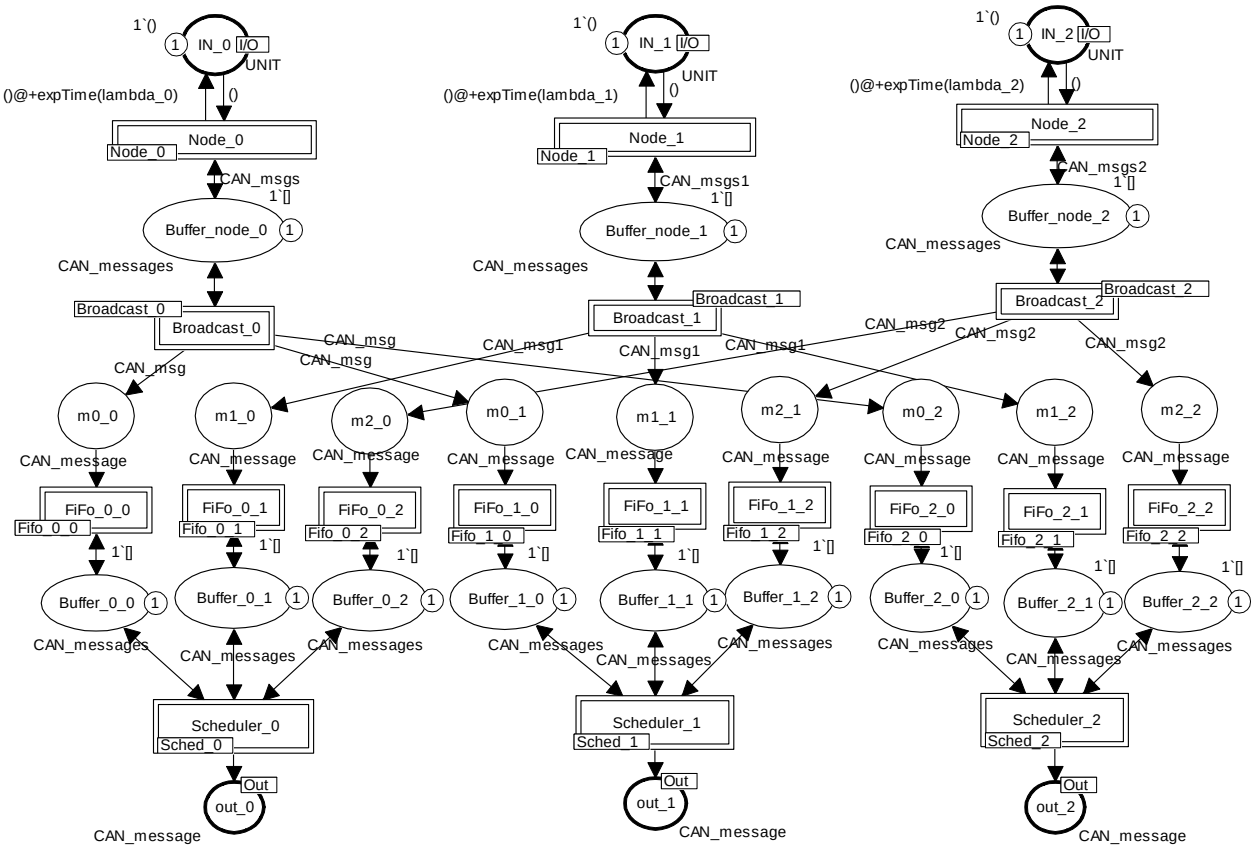


Fig. 8: SCPN Switched CAN Networks model with three nodes having three different priority classes

3.1 Generating CAN Messages (Node_i Transition)

As shown in Figure 9, the load source is modeled by the place *Next* and the transition *Generate*. Initially the place *Next* contain one token and is connected via two arcs to the transition *Generate*, the arc from *Generate* to *Next* is timed using the exponential random function. Thuswe can have a random message with parameterized inter arrival period using the value *lambda_i* (*lambda_0* for node 0: Figure 9).

The place *Buffer_node_i* is used as messages buffer, sized of 4 in our case, to decouple the source message from the Switched CAN controller. When the load source increases and no room is available in the buffer, an overflow occurs and the transition *FIFO_FULL* fires leading to lose the last generated message (due to a congestion or excessive load).

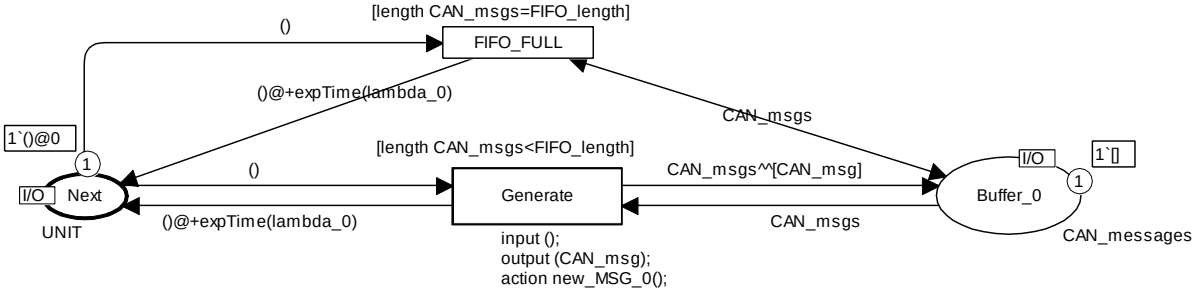


Fig. 9: Generation of CAN Message (Node_0)

3.2 Broadcasting of CAN Message (Broadcast_i Transition)

The set of broadcasting message is represented by the model described in the figure 10. Transition *Liberate* models a message coming from *Buffer_node_i*.

The *Buffer_node_i* place is a list of CAN_Message. When the list length is not null (i.e there is at least a message to send), the *liberate* Transition can be fired if the line is free (there is a token in *Line_Free* place). Otherwise, message coming from *Node_i* has to be delayed *Transfer_Data_Delay* until the previous message will be liberated. If the message is liberated, the *Server* token will be moved from the place *Line_Free* to *Line_Busy*. Then the messages will be duplicated in the right place (queues)

according to their priorities. For example, message of medium priority (*CAN_msg1*) will be routed to the places *m1_0*, *m1_1* and *m1_2*. The CAN message *m1_i* will be queued in the queue 1 of the output port i (*OUT_i*).

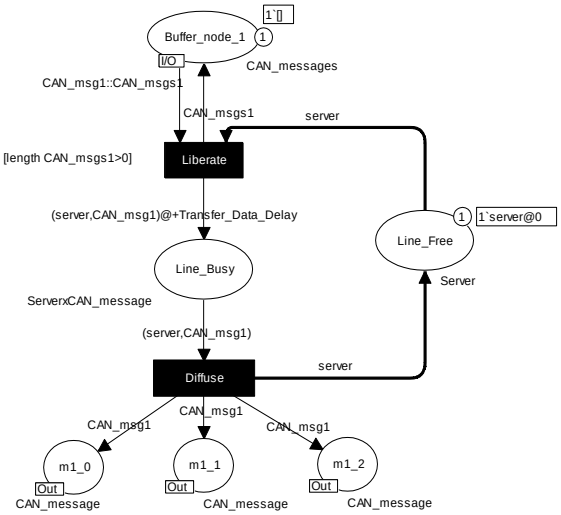


Fig. 10: Broadcasting CAN Messages generated by Node1

3.3 Storing CAN Messages (FiFo_i_j)

FIFO model is represented in the figure 11. It processes the messages in the order of their arrival. The function of the transition *Arrive* is to concatenate incoming message (*CAN_msgi*) to the *Buffer_i_j*. *Buffer_i_j* is a place having *CAN_messages* as colourset: i indicates the level of the message as it was explained previously and j indicates the output port of the model. Thus *Buffer_i_j* is the queue of Message i of the output port j (*OUT_j*). When the buffer is full (in our model *FiFo_length= 4*), the transition *Fifo_i_j_Full* is fired and the incoming message will be rejected.

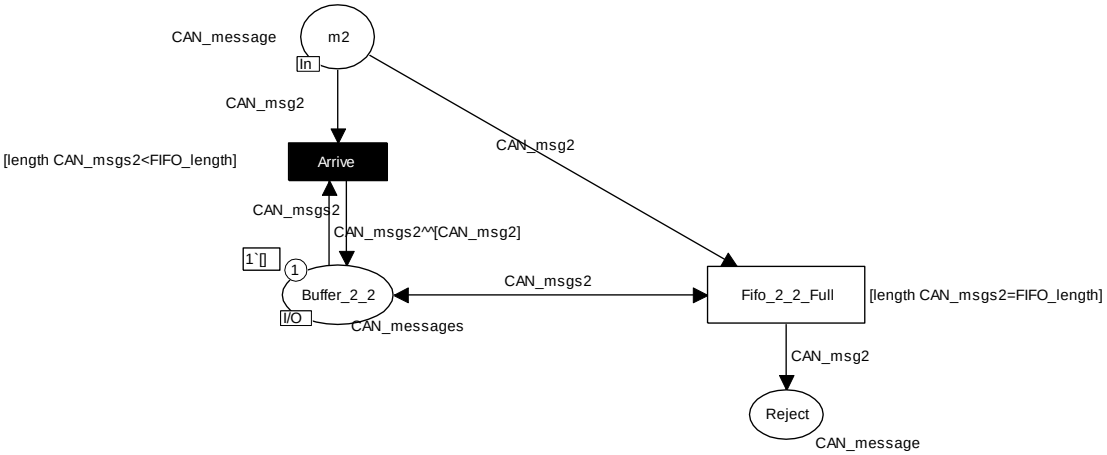


Fig. 11: Storing CAN Messages generated by Node 2 in the queue 2 of output 2

3.4 Scheduling CAN Messages (Scheduler_i)

The model of the figure 12 describes the behavior of a static priority scheduling. The type of messages is classified in three groups:

- High priority messages: These messages are generated by Node_0 and are modeled in the place *Buffer_i_0* as a list of *CAN_Message* (*CAN_msgs*) in the Scheduler of the output port i (*OUT_i*).
- Medium priority message: Those are generated by *Node_1* and are modeled in the place *Buffer_i_1* as a list of *CAN_Message* (*CAN_msgs1*) in the *Scheduler_i*.
- Low priority message; those are generated by *Node_2* and are modeled in the place *Buffer_i_2* as a list of *CAN_Message* (*CAN_msgs2*) in the *Scheduler_i*.

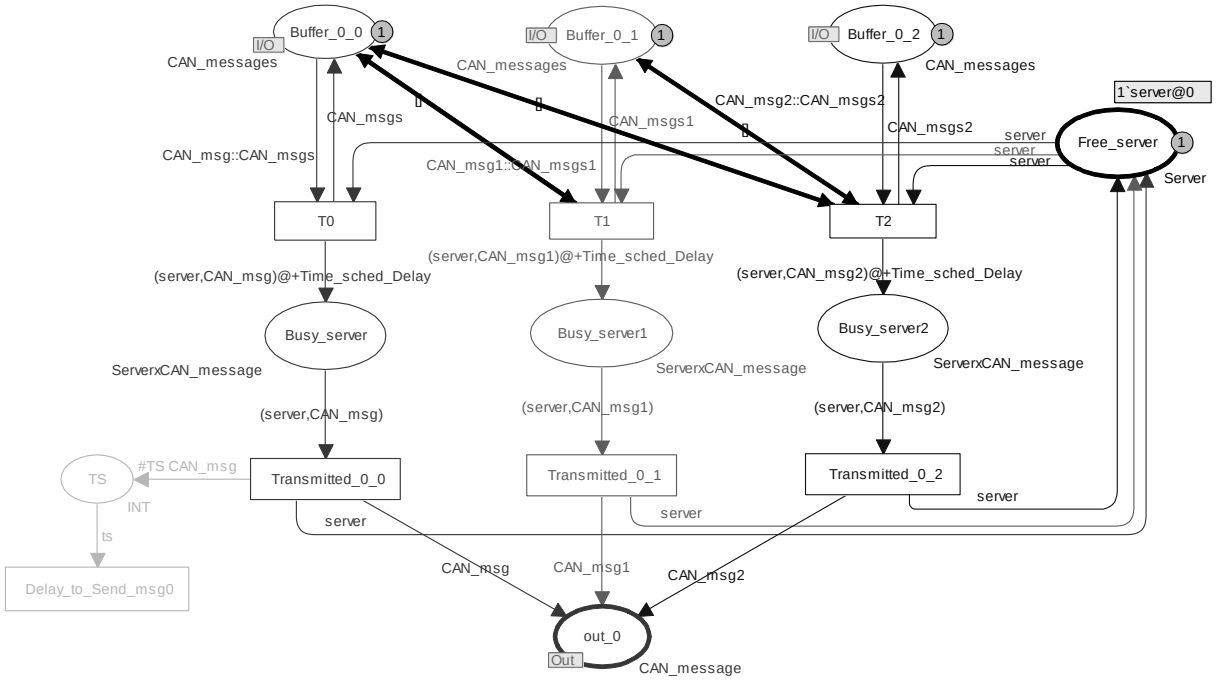


Fig. 12: Scheduling CAN Messages at the Scheduler of Output 0

A message with lowest priorities can be delayed by the other packets due to the non-preemptive characteristics of this kind of message scheduling algorithms [15].
For example, in the case of Scheduler_0, the messages of Buffer_0_2 (low priority messages) has to wait until the messages of Buffer_0_0 and Buffer_0_1 (high and medium priority messages) are fully transmitted. Then, the messages of Buffer_0_1 (medium priority messages) has to wait until the messages of Buffer_0_0 (high priority messages) are fully transmitted.

This scheduling policy is modeled using bidirectional arcs between buffers places and the transitions *T1* and *T2*. These arcs are inhibitor arcs. The method of usage of inhibitor arc is described in more details in [13].
When there is at least a message to transmit, the Ti transition can be fired if the server is free what it means that there is a token in *Free_server* place. Otherwise, the following message (according to the algorithm described above) have to be delayed *Time_Sched_Delay* until the previous message is fully transmitted. After the transmission of the message, the Server token will be moved from the place *Busy_sereri* to *Free_server*. The interest of the static priority algorithms is that it is easy to implement. Other Scheduling algorithms can be studied in future works.

4 Conclusion

In this paper, Switched Fabric CAN architecture is presented and modeled by CPNTools. The SCPN model of the Switched Fabric Controller is presented with three nodes at transmitter side using a switch fabric (3x3). For that, three message priority classes were treated with a clear representation on ID field (high, medium and low priority messages).
The model focuses on queueing, broadcasting and scheduling mechanisms which are the keys factor for the proposed architecture.
The evaluation of throughput, latency and loss probability of the proposed architecture and a comparison with Bussed CAN controller [2] will be studied in the future works to demonstrate that CAN with a crossbar switched fabric has yet a well period before its replacement.

5 References

[1] Salem Hasnaoui, Oussema Kallel "A proof-of-concept Implementation of Modified CAN Protocol on CAN Fieldbus Controller Component"; Accepted oct. 2004, Revised Jan. 2005; AMI. Journal, Ref 03/08.
[2] Oussama KALLEL, Sofiene DRIDI, Salem Hasnaoui "Modeling and Evaluating a CAN Controller Components Using Stochastic and Colored Petri Nets" IRECOS International Review on Computers and Software, Vol.4 N.1, pp 142-151; January 2009.
[3] Marko Bago, Siniša Marijan, Nedjeljko Perić; Modeling Controller Area Network Communication, proceedings of the Ninth Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, October 20-22, 2008
[4] CUI Lian-cheng, ZHAO Zheng-fang, XU Xiao-ju2, WU Fang-ming, SHAN Wei-zhen "Real Time Performance Analysis

of CAN Bus Based on TimeNET" The 3rd International Conference on Innovative Computing Information and Control (ICICIC'08), 2008.

[5] Prodanov, W.; Valle, M.; Buzas, R.; Pierscinski, H."A Mixed-Mode behavioral model of a Controller-Area-Network bus transceiver: a case study" Behavioral Modeling and Simulation Workshop, 2007. BMAS 2007. IEEE International

Volume, Issue , 20-21 Sept. 2007 Page(s):67–72

[6] Heller, C. Reichel, R. "Enabling FlexRay for avionic data buses" Digital Avionics Systems Conference DASC '09,Oct 2009.

[7] FlexRay Communication System. [Online]. available: http://www.flexray.com

[8] Brett Murphy, Emmanuel Eriksson. "Fabrics and Publish-Subscribe Schemes: A Net-Centric Blend" COTS Journal Oct. 2009 <http://www.cotsjournalonline.com/articles/print_article/100148>

[9] Rojdi Rekik, Tarek Guesmi, Salem Hasnaoui "Challenges in the Implementation, the Configuration and the Evaluation of a QoS-enabled Middleware for Real-Time Embedded Systems";

IRECOS International Review on Computer and Software vol. 3, n°3, May 2008.

[10] Arshad, Nauman, Stewart Dewar, and Ian Stalker. "Serial Switched Fabrics Enable New Military System Architectures." COTS Journal Dec. 2005 <www.cotsjournalonline.com/home/article.php?id=10043>

[11] Dr. Rajive Joshi," Using Switched Fabrics and Data Distribution Service to Develop High Performance Distributed Data-Critical Systems" The Journal of Defense Software Engineering. April 2007.

[12] K. Jensen , "Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts" (Monographs in Theoretical Computer Science, Springer-Verlag, 1997).

[13] A.V. Ratzer, L.Wells, H.M.Larsen, M.Laursen, J.F.Qvortrup, M.S.Stissing, M. Westergaard, S. Christensen, K.Jensen. CPN Tools for editing, simulating, and analysing Coloured Petri Net. LNC, 2679:450-462, 2003.

[14] Design/CPN Manuals. Meta Software Corporation and Department of Computer Science, University of Aarhus,Denmark. On-line version: http://www. daimi.aau.dk/designCPN/.

[15] John D. Pape, "Implementation of an On-chip Interconnect Using the i-SLIP Scheduling Algorithm". (The University of Texas at Austin Chap 3 pp 11-15: December 2006)

PH5WRAP: A Parallel Approach To Storage Server of Dicom Images

PDPTA

Tiago S. Soares*, Thiago C. Prado†, M.A.R Dantas‡, Douglas D.J. de Macedo§, Michael A. Bauer¶
*†‡*Informatics and Statistic Department*
Federal University of Santa Catarina
Florianópolis, Brazil
Email: steinmetz@telemedicina.ufsc.br, coeio@inf.ufsc.br, mario@inf.ufsc.br
§*Post-Graduate Program of Knowledge Engineering and Management*
Federal University of Santa Catarina
Florianópolis, Brazil
Email: macedo@inf.ufsc.br
¶*Department of Computer Sciencet*
University of Western Ontario
London, Canada
Email: bauer@uwo.ca

***Abstract*—Nowadays has been observed a large increase in the volume of images generated by medical devices. The manipulation of medical images has one known as DICOM standard that allows interoperability between these images in different devices. The server CyclopsDCMServer was developed in order to work with Hierarchical Data Format (HDF5) for manipulation of medical images (DICOM) on a distributed file system. This work aims to improve the performance of the server through an approach that uses the functions of reading and writing parallel. The experimental results indicate a gain of the proposed approach with respect to the original environment.**

***Keywords*-HDF5; MPI; PVFS; DICOM; Parallel I/O;**

I. Introduction

Databases alternative to conventional database for data storage has been a solution more sought by researchers and companies when it faces to large volumes of data in systems of high performance computing. Among the alternatives that are available and active development, currently the Hierarchical Data Format (HDF) has been highly seeking by researchers. HDF5 is the fifth version of this format, and was developed by the group "HDFGroup" from the University of Illinois in the United States. Since the release of HDF many general purpose and scientific began using it as an efficient alternative and high performance for storing and accessing large files. As examples, NASA use HDF to store data from global monitoring, clinical applications for managing large collections of images of X-rays and oil companies that store large amounts of data from 3D seismic surveys [1].

CyclopsDCMServer is a work being done by the group INCoD at Federal University of Santa Catarina in Brazil [2] and it aims to supply storage of DICOM images [3] provided by medical devices connected on *Rede Catarinense de Telemedicina* (RTCM) [4]. RTCM connects different health institutions such as hospitals and primary care that encompasses 286 cities in the state of Santa Catarina. This network provides service access to more than 10 DICOM modalities, among them are: electrocardiogram (ECG), magnetic resonance imaging (MRI), computed tomography (CT) and computed radiography (CR) [5]. For this integration occur between server and DICOM modalities, the RCTM use Picture Archiving and Communication System (PACS) [6] which includes hardware and software support for most medical equipment masking all part of communication, safety and accessibility. In Santa Catarina, the system is available almost over the state, and trend in few years cover the entire state.

A research mode of this server uses HDF5 as the data format to DICOM images and a distributed file system PVFS (Parallel Virtual File Systems), in order to focus on storage performance, DICOM image queries and solve the problem of scalability generated by storing vast amounts of files generated by PACS system. This work involves on a version of this server that includes the parallelization of the reading and writing DICOM files in HDF5. This study focuses on presents the architecture and experiments of reading and writing in parallel , with the goal of achieving better performance in access to the database and collect communication delays. This work has as motivated some parallel writing function tests performed in work [7], which demonstrated improvements in writing time.

This work is structured in six sessions. The next session begins with a base of how the CyclopsDCMServer works, and in section 3 presents the introduction of the approach of proposed architecture. In Section 4, we describe some related work. In section 5 depicts the hardware and software used in this project, the experiments and results. Finally, in Chapter 6, we ended the study with conclusions and future work.

II. CyclopsDCMServer

CyclopsDCMServer was developed in order to provide service integration of DICOM in PACS environment, in order to provide medical imaging storage and accessibility for manipulating these images through workstations. The application is multiplatform and can run on the Linux machine, Windows and other systems. Currently, in real system, the application CyclopsDCMServer stores all information provided by PACS system in a relational database called PostgreSQL. The DICOM files generated by available modalities on PACS system, generate files with different sizes, ranging from 300 Kbytes to 600Mbytes per image, and should remain stored on the server for at least 10 years. In measuring the increase in volume of data they generate problems such as scalability, latency in queries and maintenance costs.

Based on these problems described above, studies were carried out by Macedo et al [5] seeking to circumvent issues of telemedicine environments based in server with relational database systems. These issues considered are scalability, distribution of information, ability to use techniques for high system performance and operating costs. The result of this research led to a new server architecture based on PVFS and HDF5. Among the usual procedures of the current approach, the contribution was storing all the information hierarchically, such as organize and store data in HDF5 format. The second step was to use a cluster with distributed file system in order to seek high-performance disk access. Another contribution was to create an object called H5WRAP to handling with HDF5 interface.

A. H5WRAP

Due to lack of a procedure that incorporate a DICOM image in HDF5, it was necessary to create a library called H5WRAP, which converts a DICOM file in the format HDF5 data. The library contains an object that is used to create, find, collect and store information related to DICOM images in HDF5 files and this information is represented in Figure 1. Among various metadata contained in a DICOM file [8], for the H5WRAP, were collected only metadata that are importance to the PACS system used in Santa Catarina.

HDF5 files are organized in a hierarchical structure. It is observed that were selected three DICOM layers, each layer represent a group of study, serie and image, where the hospital attributes and patient studies are contained in the study layer. Their primary structures are two datasets and groups [1] . An HDF5 group is a grouping structure that can contain zero or more instances of groups or datasets, while an HDF5 dataset is a multidimensional array of data and also contains metadata that describes the dataset. In the next section, we explain how the read and write in parallel was built in H5WRAP

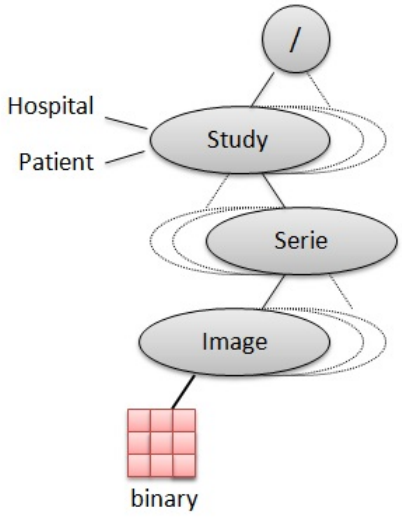


Figure 1. DICOM layers in HDF format

B. PH5WRAP

The PH5WRAP is an architecture oriented to work in reading and writing parallel only the binary. The reason for this restriction is due to the fact that the binary image representing on average 90% of a DICOM file, ignoring the need to parallelize small disk accesses to metadata, which could result in a longer process, due to communication necessary to distributed these layers. This architecture is designed to work in the same environment previously proposed in H5WRAP, based in PVFS and HDF5, with the addition of the Message Passing Interface (MPI).

The parallel access is provided by processes that are initiated at each node. The parallel access comes from the HDF5 API, which uses ROMIO interface to access the file system. ROMIO is a portable implementation of MPI-IO access which was created to provide high performance access to distributed file system. Its development is restricted to support some types of file system and among the most widely used, we have the PVFS and NFS [9]. In terms of characteristics of parallel access, the Parallel Virtual File Systems designed to support multiple access models, such as collective and independent access, as well as non-contiguous access patterns and structured. PVFS has three important structures: the metadata node, data node and client node. A node can represent all the structures simultaneously. The metadata nodes are controllers of permission file, directories and file names. Data nodes store physically data files and the client makes requests to the file system commands using POSIX or through APIs .

The Figure 2 illustrates the architecture of PH5WRAP, which illustrates the communication between the usual H5WRAP with the parallel application. The implementation of parallel application will be used by the server as it needs to perform a read operation or write a binary image DICOM.

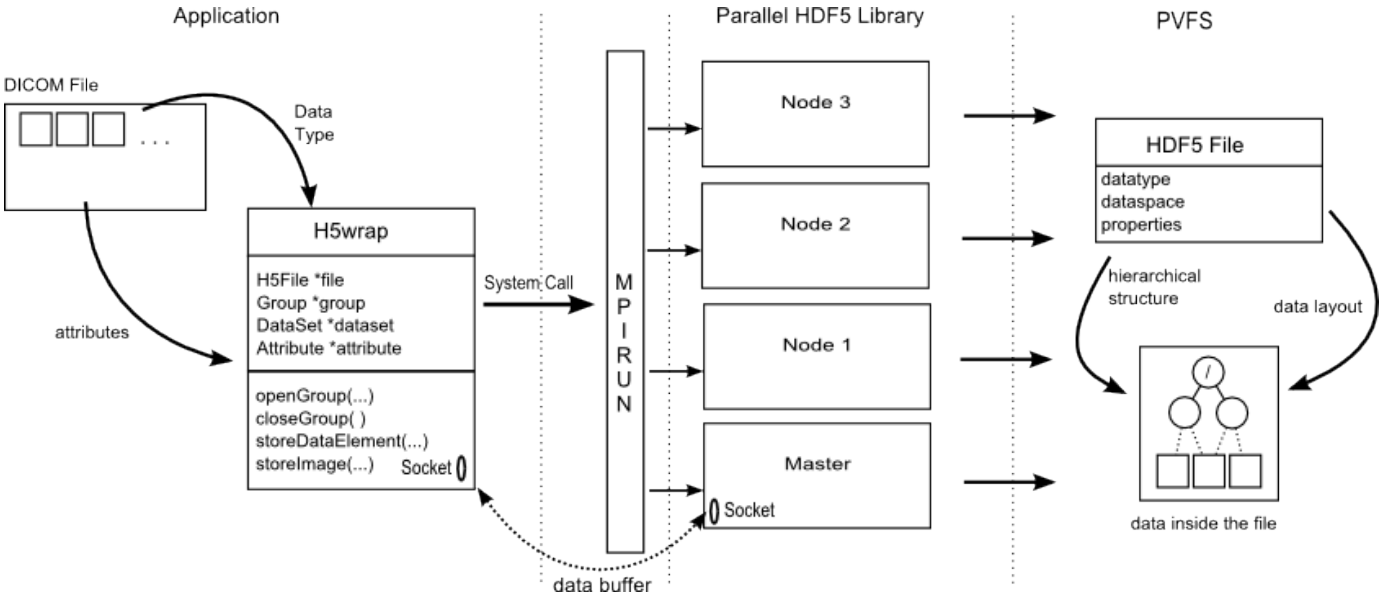


Figure 2. Ph5wrap architecture

Basically, the dicom file is unpacked to the data type and attributes, wrapping these contents into HDF5 format and throwing the binary image to parallel application. In this application, two main procedures are performed, the reading and writing functions. In case of Figure 3, it illustrates the division of binary image between the application processes when the master node receives the buffer from H5wrap. The division it's make simple, based on number of parallel nodes, which more nodes, smaller the buffer to distributed to other nodes.

The next two sections, it's describe the functionality of the both main function.

1) Write function: The write function begins with a request for some DICOM modality available on the PACS for storage an image in CyclopsDCMServer. As the object H5WRAP performs file analysis , it is create indexes in Clucene index table [10]. Before writing the metadata, it is checked whether the study groups, series and image metadata are already available in the HDF5 file, otherwise it is create the layers. Stored metadata, the server creates a connection through socket with the master process, sending the type of operation, the operation path and binary buffer, awaiting return the status of communication process. The Master process distributes the binary partition for nodes according to number of nodes in the system, and they store their respective buffer in system PVFS.

2) Read function: The process of reading function is similar to the writing process. Start with a client application consulting an image. This query is performed on server via Clucene index table where DICOMS images can be filtered by modalities study, series and images. The retrieve for an image is performed after the step above, that returns the

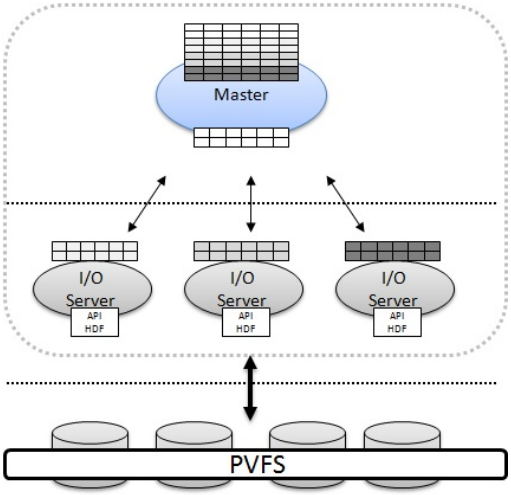


Figure 3. Shows the binary buffer distribution between nodes

object path that will be accessed in HDF5 file. The reading of metadata is performed by H5WRAP, while the binary image is performed by the parallel application. To performing the reading of the binary, H5WRAP create connection with master process, sending only the type of operation and location of the binary access. The Master node distributes to other nodes the location, which each node retrieves a region of binary. After reading their region, the nodes return a binary portion to master process. The Master, taken up all the regions, it sends the binary to H5WRAP. The H5WRAP ends the DICOM file, returning to the client.

III. Related Work

The first two works are themes that seek to resolve I/O performance issue of large amounts of stored data, using the parallel HDF5 library. The first related work [11], is very similar to ours where this work focus on the use of parallel I/O library for scalability problems involving many fermions dynamics for nuclear structure (MFDn). This study used a parallel version of HDF5 and performed tests on the independent and collective models. The results showed that files with size up to 20 GB, parallel HDF5 became more productive than sequential. The work [12] focuses in using parallel-IO for particle-based accelerator simulations that involved large amounts of data and dimensional arrays. They used two different API, the first is well-known MPI-IO and the second is the parallel HDF5 called H5part. He compared the simulations performance of read and write file operation between the API H5part, MPI-IO and the primary (file per process). HDF5 showed good performance in writing, although MPI-IO showed better results in both operations.

The works [13] [14] [15] are interesting because they focused in working with comparisons and performance improvements of I/O in platforms such as Jaguar and Red Storm. These are supercomputing platforms, which provide computer services, such as GTC for spectrum, Parallel Ocean Program (POP) ocean modeling, physics calculations program and others. The work proposed by Yu et al has some interesting comparisons with different types of I/O. They propose to improve and enhance access in a parallel application called Jaguar of Oak Ridge National Laboratory and aimed at several objectives such as, the characterization of performance parallel-IO on storage unit and scalability of the entire system. The results showed a better performance and less scalability in parallel-IO in collective mode and showed that using technique called FLASH I/O, the performance can improve by 34% with certain adjustments on collective I/O .

About the file systems, the work [16] conducted three experiments in an atmospheric model system, distinguishing the access mode on disk. The experiments deal with applying a single application with a differential in system architecture that adds Threads on its processes. The written tests with local disk had the best performance by not requiring communication between the nodes of the file system. The environment with threads had a better performance when used the parallel virtual file system (PVFS), because it has facilities when use MPI interface. The experiments were performed using 30 clients in one PVFS environment with four nodes. As each process accesses the file system constantly to write the atmospheric results, when you have more than one process per machine, they compete for access disk, reducing the execution time. This does not occur with the application threads, since it runs one process per machine and performs better memory used.

IV. Environment And Experimental Results

The environment used for the experiments is a virtualized with eight virtual machines, both PVFS nodes and nodes of the parallel application. Another virtual machine is used to host the server CyclopsDCMServer. For virtualization platform environment, we used VMware vSphere 5. The host machine has an Intel Xeon E5310 with 4 cores at 1.6 GHz each and a total of 10Gbytes of memory and 460 Gbytes of disk. Each virtual machine has virtualized 1077 Gbytes of memory and 23:26 Gbyte disk.

To assemble the virtual cluster, it was necessary to set up an environment with PVFS, MPICH2and HDF5, and their arguments are meant for installation work with parallel features. Each of the parallel application process will also be a PVFS data storage node. We use the same number of PVFS data nodes as parallel application processes in order to use the maximum use of the environment. The graphs bellow shows the number of attempts on y axis and time spent in seconds on x axis to complete the experiments.

In reading experiments were total conducted 24 experiments, 16 collecting only the read time of binary from two DICOM images in HDF file, discarding the communication delay and assembly of the DICOM file. The other eight experiments represent the total value of reading the images, plus the assembly of the DICOM file. The first binary image has 92.16Mbytes while the second has 52.42 Mbytes. Experiments were performed with the serial server, three more experiments with 2, 4 and 8 parallel nodes. These experiments were performed twice, resulting in a total of eight experiments per image. Each reading experiment was repeated 25 times on the same image. From these results, it is collected the average and presented in the comparison chart of the averages in Figure 4.

It can be observed in the first average graph of image 1 on the first attempt, a difference up to 6 milliseconds of average serial reading with 4 parallel nodes. In the second attempt, the parallel reading with 8 nodes have underperformed with a difference of approximately 7 milliseconds compared to the serial reading and near twice worse compared with 4 nodes. The same can be observed in image 2, although the results are better than eight nodes serial reading, the difference between the applications with four nodes is almost twice. Finally, the Figura 6 graph is a noteworthy the impact of the runtime application when used CyclopsDCMServer with 8 nodes.

One important factor that causes the loss of performance of parallel application with 8 nodes in the three graphs is due to be working with one processor with four cores and using 8 vitual machines, running one process. This process will disputed to use one core, therefore, others have to wait. This causes saturation in processor use, rather than working with a virtual machine requires a high processing power. This

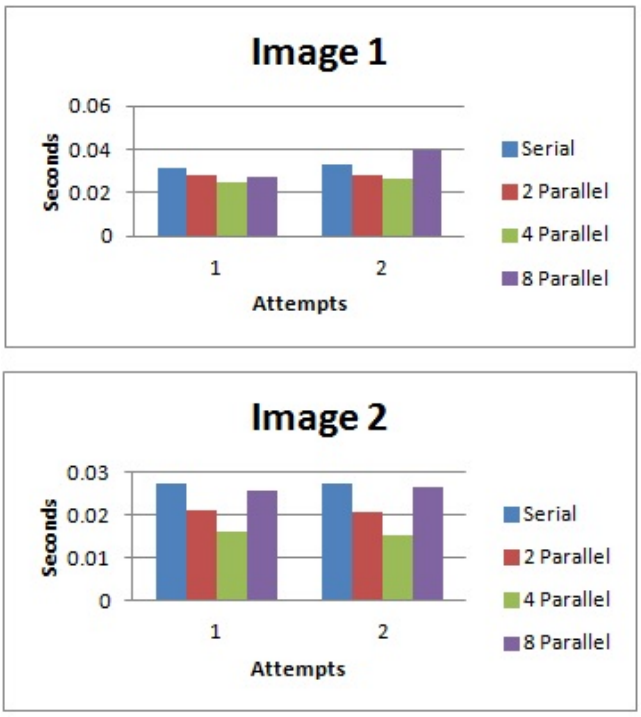


Figure 4. Reading comparisons graphics

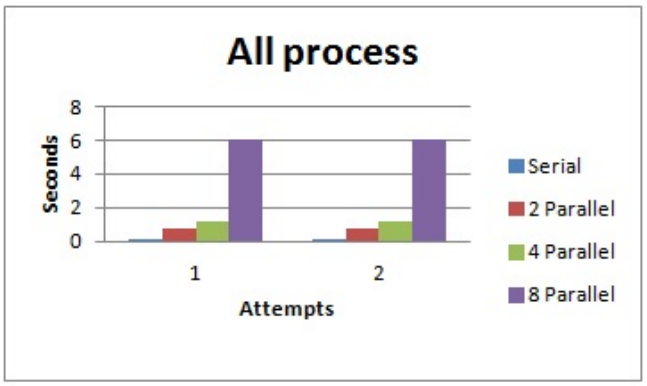


Figure 5. Server runtime

conclusion can be proved also by the fact that when we use only four nodes, no process have to wait, the performance is approximately 2 times faster. This underperformance provide by 8 nodes is also observed in Figures 5.

The next figures illustrate two graphs of average for written experiments performed with about 1000 DICOM images each attempts. These images are sent to the server CyclopsDCMServer, which is responsible to communicate with parallel application.

In Figure 6 shows the average time of writing an image in the HDF. For this first graph, we collected the time of writing only, discarding the time of the system. Among the two attempts, there is the bottleneck described above

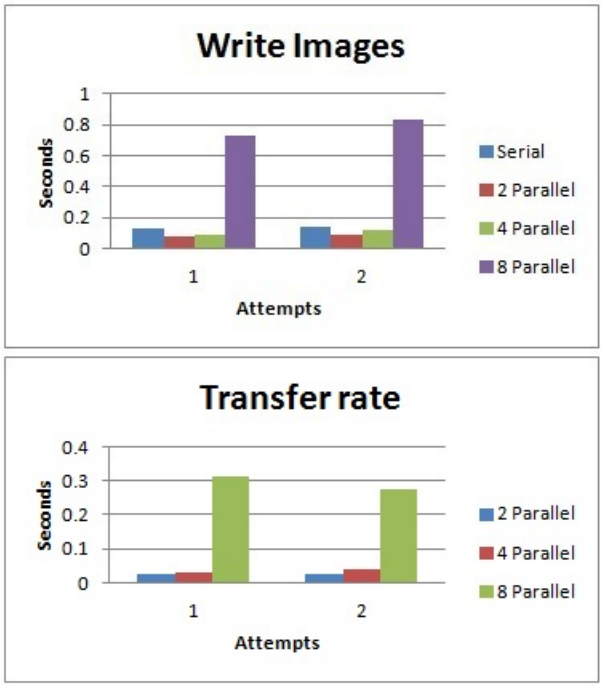


Figure 6. Written comparisons graphics and graph of transfer rate

about the 8 nodes and performance gain of 2 and 4 nodes. This gain is around 50 milliseconds and 33 respectively compared with serial mode. The transfer rate is shown in the second graph. This transfer rate is the measurement of the binary transmission from the server to the MASTER of the parallel application. Unlike the total time of the system, the transfer rate does not take into account other communications between the MASTER and the server, as well as communication between the application processes. Note that the speed of data communication is directly interconnected to the number of processes in the system.

V. Conclusion And Future Work

This work was done in order to show new results for the parallel architecture introduced earlier in [7], in order to detect failures and performance in reading and writing parallel of a DICOM image in HDF format. The environment is virtualized and configured with parallel virtual file system (PVFS) and HDF5, where more tests were realized to collect more consistent results.

The results showed a gain in reading and writing parallel, but the number of nodes exceeds the number of processing from the host machine, the performance of the architecture is low. Other problem in this system is the poor time of runtime system, provided by the communication between the server and parallel approach to transfer binary buffer. One solution to overcome the time problem is to move the H5wrap object to the master process, with goal to remove the necessity of binary transfer.

As future work, we intend that run parallel architecture in a real environment and dedicated to collect results that do not depend mainly on the processing power. Analyze the performance of system access in accordance with increasing the number of nodes. Finally, more tests with different types of DICOM modalities, mainly work with larger files.

REFERENCES

[1] HDFGroup, Access: January, January 2011. [Online]. Available: http://www.hdfgroup.org

[2] "Romio: A high-performance, portable mpi-io implementation," Access: December. 2010, December. [Online]. Available: http://cyclops.telemedicina.ufsc.br

[3] O. Pianykh, *Digital Imaging and Communications in Medicine (DICOM): A practical introduction and survival guide*. Springer Verlag, 2008.

[4] J. Wallauer, A. von Wangenheim, R. Andrade, and D. De Macedo, "A telemedicine network using secure techniques and intelligent user access control," in *21st IEEE International Symposium on Computer-Based Medical Systems*. IEEE, 2008, pp. 105–107.

[5] D. De Macedo, A. Von Wangenheim, M. Dantas, and H. Perantunes, "An architecture for dicom medical images storage and retrieval adopting distributed file systems," *International Journal of High Performance Systems Architecture*, vol. 2, no. 2, pp. 99–106, 2009.

[6] T. Cradduck, "Pacs: Basic principles and applications," *Physics in Medicine and Biology*, vol. 45, p. 2444, 2000.

[7] T. Soares, D. de Macedo, M. Bauer, and M. Dantas, "A parallel architecture using hdf for storing dicom medical images on distributed file systems."

[8] "Digital imaging and communications in medicine," Access: December. 2011, December. [Online]. Available: http://en.wikipedia.org/wiki/DICOM

[9] M. Division, "Romio: A high-performance, portable mpi-io implementation," Access: March. 2011. [Online]. Available: http://www.mcs.anl.gov/research/projects/romio/

[10] Sourceforge, "Clucene," Access: December, December 2011. [Online]. Available: http://clucene.sourceforge.net/

[11] N. Laghave, M. Sosonkina, P. Maris, and J. Vary, "Benefits of parallel i/o in ab initio nuclear physics calculations," *Computational Science–ICCS 2009*, pp. 84–93, 2009.

[12] A. Adelmann, R. Ryne, J. Shalf, and C. Siegerist, "H5part: A portable high performance parallel data interface for particle simulations," in *Particle Accelerator Conference, 2005. PAC 2005. Proceedings of the*. IEEE, 2006, pp. 4129–4131.

[13] M. Fahey, J. Larkin, and J. Adams, "I/o performance on a massively parallel cray xt3/xt4," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[14] J. Laros, L. Ward, R. Klundt, S. Kelly, J. Tomkins, and B. Kellogg, "Red storm io performance analysis," in *Cluster Computing, 2007 IEEE International Conference on*. IEEE, 2007, pp. 50–57.

[15] W. Yu, J. Vetter, and H. Oral, "Performance characterization and optimization of parallel i/o on the cray xt," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–11.

[16] F. Boito, R. Kassick, L. Pilla, N. Barbieri, C. Schepke, P. Navaux, N. Maillard, Y. Denneulin, C. Osthoff, P. Grunmann *et al.*, "I/o performance of a large atmospheric model using pvfs," *Rencontres francophones du Parallélisme (RenPar20)*, 2011.

Optimizing The Locking Methods in Distributed Database Systems

Mehdi Assefi
Dept. of Computer Engineering
Islamic Azad University, Neyshabur Branch
mhd_asefi@msn.com

Abstract

The main objective of this paper is to analyze the results of evaluation on different capabilities of concurrency control in Distributed databases, and examine different choices for locking and required settings on a sample database.

We have chosen the Oracle database of a paperless filing system as the basis for examining. In this distributed database, different transactions could access to a database through different stations and implement some basic operation on it. As we could assume, using a proper mechanism for concurrency control seems to be necessary. By evaluating possible choices and effect of their performance on the throughput of the system, and mean response time of transactions, we will try to evaluate pro and con for each method.

1. INTRODUCTION

As mentioned earlier, Oracle enables the developers to change the architecture of the database using the programming, and at the background. Host languages will give this chance to developers, by making connection with Oracle. using a proper mechanism for concurrency control seems to be necessary. By evaluating possible choices and effect of their performance on the throughput of the system, and mean response time of transactions, we will try to evaluate pro and con for each method.

2. Manual setting at the background

2.1. Using PL/SQL and Java

As you know, you can use a structural control for grouping SQL commands in a PL/SL block and then send the block to the Oracle.
it is also possible to use PL/SQL and Java procedures and reduce calling of the database in the program. For example, for running one SQL command, 10 calling will be needed, but for running a micro program containing 10 SQL commands, only one calling will be needed. This will reduce the network traffic. We can compile PL/SQL and Java procedures separately and put in a database. These procedures will be passed to PL/SQL motor after calling. Moreover, only one copy of these procedures must be loaded in the memory for being used by multiple user.

PL/SQL can also cooperate with Oracle developing tools such as Oracle Forums and Oracle Reports. By adding processing power, performance will increase. Using PL/SQL will cause calculation to be run more efficient and fast and without being called within Oracle. Result will be reduction of response time and network traffic.

2.2. Using advantages of row locking

Oracle enables databases to lock row of data and avoid locking the whole table. Row locking will make increasing of concurrency possible and many users will be able to access to the rows of a table. By this, performance will increase sharply.

2.3. Configuring parameters of a database

Using the host languages and applicable parameters, we are able to optimize the structure of a database. We hereby mention some of these parameters.

db_block_size

This parameter defines the size of blocks in the database. As we know, block is the scale for transferring information from the disk. By increasing the size of the block, we will have this chance to read all items that are at the same class. This will reduce the mean time needed by disk for accessing a unit of data. This parameter will be defined when we create the database and will be permanent. Precise identifying the size will be useful for preventing the information to become scrappy.`

db_cache_size

This parameter will identify the size of data buffer. Data will be read from disk and will be placed in this buffer in order to minimize access request to the disk. Results show that irregular increase of this buffer will drastically reduce the performance, which would be due to the extra burden posed on the operating system for buffer management and reducing the memory (result of occupying the buffer by data).

db_file_multiblock_read_count

This parameter identifies number of block which will be read in any referral to the disk.

Java_pool_size

In case if a database uses Java codes, proper set up for this section will have positive effect on the performance. This place creates a location in the buffer for putting Aplet Java codes.

large_pool_size

If database is supposed to transfer huge blocks, proper set up for this section will be very helpful. These objects will be considered when online backup is undertaken and object is loaded in the memory. Hereby list of these parameters for our database follows. We will change the parameters to evaluate the results.

Cache and I/O

db_block_size=8192

db_cache_size=25165824

db_file_multiblock_read_count=16

Cursors and Library Cache

open_cursors=300

Database Identification

db_domain=""

db_name=ASY_DB

Diagnostics and Statistics

background_dump_dest=G:\oracle\admin\myora\bdump

core_dump_dest=G:\oracle\admin\myora\cdump

timed_statistics=True

user_dump_dest=G:\oracle\admin\myora\dump

File Configuration

Control_files=("G:\oracle\oradata\ASY_DB\ctrl1ASY_DB.ctl",

"G:\oracle\oradata\ASY_DB\ctrl3ASY_DB.ctl")

Instance Identification

Instance_name=ASY_DB

Miscellaneous

Compatible=9.2.0.0.0

Optimizer

Hash_join_enabled=TRUE

Query_rewriter_enabled=FALSE

Star_transformation_enabled=FALSE

Faster_statr_mttr_target=300

Security and Auditing

Remote_login_passwordfile=EXCLUSIVE

Sort, Hash Joins, Bitmap Indexes

Pga_aggregate_target=25162824\sort_area_size=524288

System Managed Undo and Rollback Segments

Undo_management=AUTO

Undo_retention=10800

Undo_tablespace=UNDOTBS1

3. Optimizing I/O parameters related to data involvement

In order to see the change in the events related to I/O, in any system we can use perfmon. By observing counters related to the system parameters, we could have required changes. Following parameters are available:

Disk reads/Sec	**number of reads per second**
Disk write / sec	**number of writes per second**
Disk transfer / sec	**number of transfers per second**
Avg. Disc Sec/Read	**average time spent for reading**
Avg. Disc Sec/Write	**average time spent for writing**
Avg Disc Sec/Transfer	**average time spent for reading and writing**
Avg Disk Queue/Length	**average number of I/O in I/O sub system**

For example, in the system that we analyze, I/O operations will be complete in 20-30 ms. If this amount increase from 6 ms for a disk, system will halt and database must be optimized.

3.1. Using UTLBSTAT and UTLESTAT

To useful parameters within Oracle are UTLBSTAT and UTLESTAT. There are some documents inside ORANT\RDBMS80\ADMIN folder that use internally by Oracle. When the database is going to be created, CATPROC.ORA and CATALOG.ORA will run from this folder. UTLBSTAT and UTLESTAT also run from this folder.

UTLBSTAT sets up internal tables and creates an instant photo from Oracle's internal counter. It will then create another photo after a high loaded operation by UTLESTAT and compares the results. UTLESTAT introduces number of complete counter values and a full statistical briefing during this period. It must be considered that UTLBSTAT.SQL and UTLESTAT.SQL both have a CONNECT INTERNAL string at top. Obviously both of these documents will have problem in NT. This row must be marked as explanation. For doing this, we put Rem at the beginning of them. UTLESTAT and UTLBSTAT could be run using SVRMGR30 application. After being connected to the database as INTERNA or SYS we can run UTLBSTAT as follow:

D:\ORANT\RDBMS80\ADMIN\UTLBSTT;

After a heavy load operation and after a considerable amount of time UTLESTAT must be implemented using the following pattern:

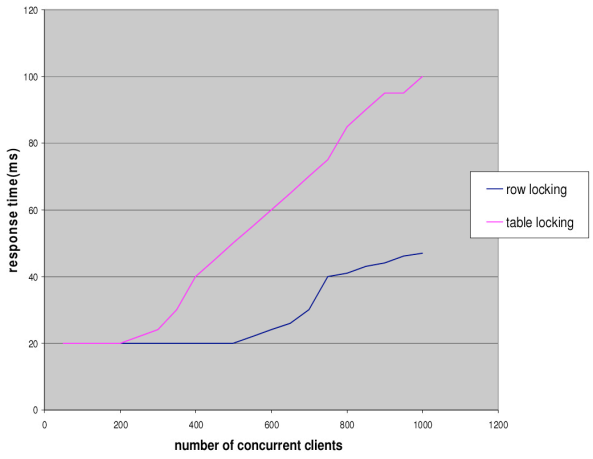
D:\ORANT\RDBMS80\ADMI\UTLESTAT;

4. Evaluating the performance

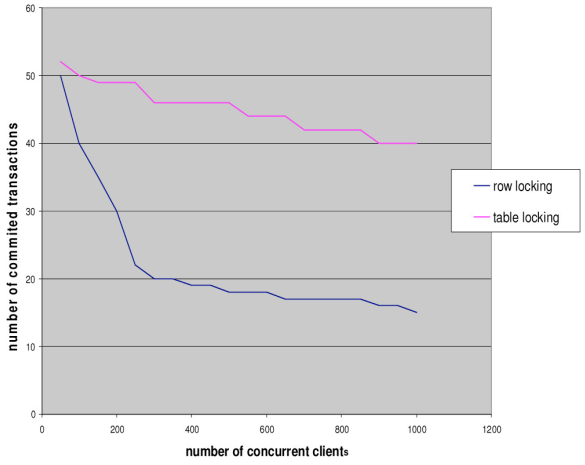
We now want to analyze the implementing time of our database's transaction considering different parameters. For increasing the work content and enhancing the workload, we will run transactions with increasing content on the database. We will then calculate the response time of the transactions, using the introduced calculation tools.

We first apply table locking rather than row locking. Results show that row locking will reduce the time that transactions need to access data, and in result response time will decrease and operational capability will increase. Chart related to the effect of these changes is depicted in the chart below.

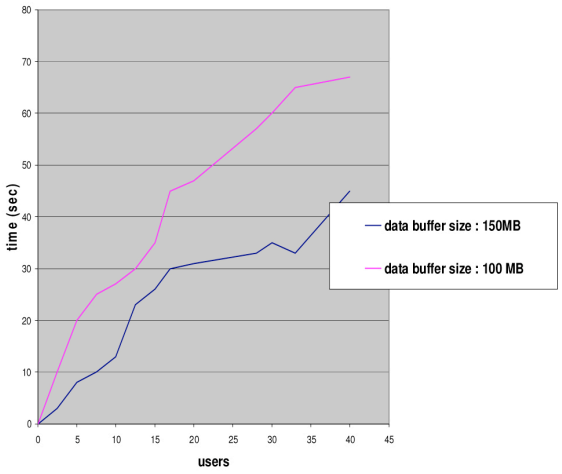
We use 100 MB and 150 MB buffers. Database volumes in both cases were 1.6 GB. We used IMPORT/EXPORT and INSERT commands in this evaluation. Number of clients was vary from 10 to 40. Results show that using a bigger buffer will result in reduction of the time needed by transactions to access the data, wich will reduce the operation time and will increase operational capability. Following charts will depict the effect of these changes.



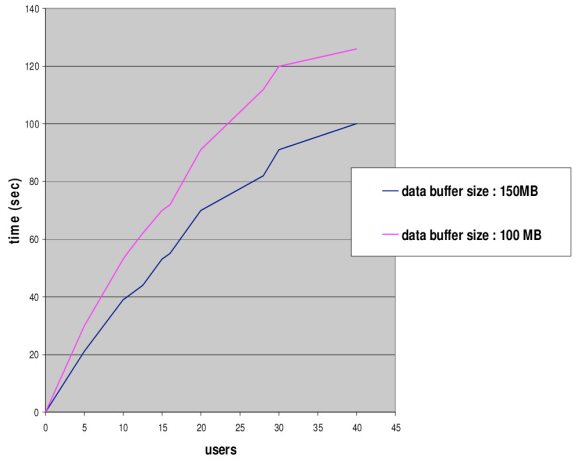
Effect of using row locking on transaction's response time



Effect of using row locking on throughput of system



Effect of using a bigger locking queue on response time of INSERT transaction



Effect of using a bigger locking queue on response time of IMP/EXP transactions

5. Summary

Oracle database is capable of doing manual setting using the programming. Majority of operation related to managing concurrency control is possible to be undertaken manually. The focus will be on increase or decrease on the length of queues and data buffers, and row locking and table locking. By considering system specifications ad proper adjustment for these parameters and other parameters of I/O, we will be able to considerably increase the performance of the database.

REFERENCES

[1] PETER M. G. APERS, ALAN R. HEVNER, S. BING YAO: OPTIMIZATION ALGORITHMS FOR DISTRIBUTED QUERIES. IEEE TRANS. SOFTWARE ENG. 9(1): 57-68(1983).

[2[PHILIP A. BERNSTEIN , NATHAN GOODMAN , EUGENE WONG , CHRISTOPHER L. REEVE , JAMES B. ROTHNIE, JR., QUERY PROCESSING IN A SYSTEM FOR DISTRIBUTED DATABASES (SDD-1), ACM TRANSACTIONS ON DATABASE SYSTEMS (TODS), V.6 N.4, P.602-625, DEC. 1981 [DOI>10.1145/319628.319650]

[3] PHILIP A. BERNSTEIN , DAH-MING W. CHIU, USING SEMI-JOINS TO SOLVE RELATIONAL QUERIES, JOURNAL OF THE ACM (JACM), V.28 N.1, P.25-40, JAN. 1981 [DOI>10.1145/322234.322238]

[4] P.A. BLACK AND W.S. LUK. A NEW HUERISTIC FOR GENERATING SEMI-JOIN PROGRAMS FOR DISTRIBUTED QUERY PROCESSING. PROC. IEEE COMPSAC, DECEMBER, 1982, PP. 581-588.

[5] A.L.P. CHEN AND V.O.K. LI. PROPERTIES OF OPTIMAL SEMI-JOIN PROGRAMS FOR DISTRIBUTED QUERY PROCESSING. PROC. IEEE COMPSAC, NOVEMBER, 1983, PP. 476-483.

[6] A.L.P. CHEN AND V.O.K. LI. DERIVING OPTIMAL SEMI-JOIN PROGRAMS FOR DISTRIBUTED QUERY PROCESSING. PROC. IEEE INFOCOM, APRIL, 1984.

[7] A.L.P. CHEN AND V.O.K. LI. IMPROVING SEMI-JOIN PROGRAMS FOR DISTRIBUTED GUERY PROCESSING. TO APPEAR IN PROC. IEEE COMPSAC, NOVEMBER, 1984.

[8] TO-YAT CHEUNG, A METHOD FOR EQUIJOIN QUERIES IN DISTRIBUTED RELATIONAL DATABASES, IEEE TRANSACTIONS ON COMPUTERS, V.31 N.8, P.746-751, AUGUST 1982 [DOI>10.1109/TC.1982.1676081]

[9] D.M. CHIU. OPTIMAL QUERY INTERPRETATION FOR DISTRIBUTED DATABASES. PH.D TH., HARVARD UNIVERSITY, DECEMBER 1979.

[10] DAH-MING CHIU , PHILIP A. BERNSTEIN , YU-CHI HO, OPTIMIZING CHAIN QUERIES IN A DISTRIBUTED DATABASE SYSTEM., SIAM JOURNAL ON COMPUTING, V.13 N.1, P.116-134, FEB. 1984 [DOI>10.1137/0213009]

[11] E. F. CODD, A RELATIONAL MODEL OF DATA FOR LARGE SHARED DATA BANKS, COMMUNICATIONS OF THE ACM, V.13 N.6, P.377-387, JUNE 1970 [DOI>10.1145/362384.362685]

[12] E. F. CODD: RELATIONAL COMPLETENESS OF DATA BASE SUBLANGUAGES. IN: R. RUSTIN (ED.): DATABASE SYSTEMS: 65-98, PRENTICE HALL AND IBM RESEARCH REPORT RJ 987, SAN JOSE, CALIFORNIA : (1972).

[13] ROBERT S. EPSTEIN, MICHAEL STONEBRAKER: ANALYSIS OF DISTRIBUTED DATA BASE PROCESSING STRATEGIES. VLDB 1980: 92- 101.

[14] G.D. HELD, M.R. STONEBRAKER AND E. WONG. INGRES - A RELATIONAL DATA BASE SYSTEM. PROC. NCC, 1975.

[15] A. HEVNER, QUERY OPTIMIZATION IN DISTRIBUTED DATABASE SYSTEMS. PH.D. TH., U. OF MINNESOTA, 1979.

[16] ALAN R. HEVNER, S. BING YAO: QUERY PROCESSING IN DISTRIBUTED DATABASE SYSTEMS. IEEE TRANS. SOFTWARE ENG. 5(3): 177-187(1979).

[17] K.T. HUANG, QUERY OPTIMIZATION IN DISTRIBUTED DATABASES. PH.D. TH., M.I.T., DECEMBER 1982.

[18] W. S. LUK, LYDIA LUK: OPTIMIZING SEMI-JOIN PROGRAMS FOR DISTRIBUTED QUERY PROCESSING. ICOD 1983: 298-316.

[19] SUGIHARA, K., ET. AL. OPTIMIZATION ALGORITHMS FOR PROCESSING SIMPLE QUERIES IN STAR NETWORKS. PROC. IEEE COMPSAC, NOVEMBER, 1983, PP. 547-554.

[20] CLEMENT T. YU, K. LAM, C. C. CHANG, S. K. CHANG: PROMISING APPROACH TO DISTRIBUTED QUERY PROCESSING. BERKELEY WORKSHOP 1982: 363-390.

[21] C. T. YU , C. C. CHANG, ON THE DESIGN OF A QUERY PROCESSING STRATEGY IN A DISTRIBUTED DATABASE ENVIRONMENT, PROCEEDINGS OF THE 1983 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, MAY 23-26, 1983, SAN JOSE, CALIFORNIA [DOI>10.1145/582192.582203]

Hybrid Single/Double Precision Floating-Point Computation on GPU Accelerators for 2-D FDTD

Hasitha Muthumala Waidyasooriya, Yasuhiro Takei, Masanori Hariyama and Michitaka Kameyama
Graduate School of Information Sciences, Tohoku University
Aoba 6-6-05, Aramaki, Aoba, Sendai, Miyagi, 980-8579, Japan
Email: {hasitha, takei, hariyama, kameyama}@ecei.tohoku.ac.jp

Abstract—*Acceleration of FDTD (Finite-Difference Time-Domain) is very important in computational electromagnetic. We propose a hybrid single/double precision floating-point computation to accelerate FDTD on GPUs. We apply single-precision when the dynamic range of the electromagnetic field is low and double-precision when the dynamic range is high. According to the experimental results, we achieved over 35 times of speed-up compared to the CPU implementation and over 1.79 times speed-up compared to the conventional GPU acceleration.*

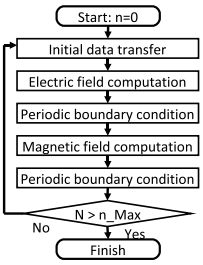
Keywords: GPGPU, FDTD, high-performance-computing

1. Introduction

Computational electromagnetic shows a rapid development recently due to the introduction of processors that have parallel processing capability such as multicore CPUs and GPUs (Graphic Processing Units). FDTD (Finite-Difference Time-Domain) algorithm [1] is one of the most popular method of computational electromagnetic simulation due to its simplicity and very high computational efficiency. It has been widely used in many applications such as coil modeling [2], resonance characteristics analysis [3], etc. Many of these applications require double precision floating-point computation to satisfy the stability condition [4].

There are many works that use GPUs [5] to accelerate FDTD. Such works consider how to parallelize the FDTD computation so as to use many nodes as possible. However, using more nodes means more cost and more power consumption. In this paper, we focus on extracting more performance from the same hardware by using high-precision computation only when it is necessary.

We consider the FDTD method used in resonance characteristics analysis of a cylindrical cavity [3] as the example application. We analyze the characteristics of the application and apply double-precision floating-point computation when the dynamic range is large and single-precision floating-point computation when the dynamic range is small. dynamic range refers to the ratio between the largest and the smallest values of electric (or magnetic) field. According to the experimental results, we achieved over 35 times of speed-up compared to the CPU implementation. This speed-up is almost 1.79 times of the conventional GPU acceleration.



(a) FDTD process

$$E_z^{n+1}(i,j) = E_z^n(i,j) - P_y(i,j)\left\{H_x^{n+\frac{1}{2}}(i,j+1/2) - H_x^{n+\frac{1}{2}}(i,j-1/2)\right\}$$
$$+ P_x(i,j)\left\{H_y^{n+\frac{1}{2}}(i+1/2,j) - H_y^{n+\frac{1}{2}}(i-1/2,j)\right\}$$
$$H_x^{n+\frac{1}{2}}(i,j+1/2) = H_x^{n-\frac{1}{2}}(i,j+1/2) - Q_y(i,j)\left\{E_z^{n+1}(i,j+1) - E_z^n(i,j)\right\}$$
$$H_y^{n+\frac{1}{2}}(i+1/2,j) = H_y^{n-\frac{1}{2}}(i+1/2,j) - Q_x(i,j)\left\{E_z^{n+1}(i+1,j) - E_z^n(i,j)\right\}$$

(b) FDTD computation

Fig. 1: FDTD

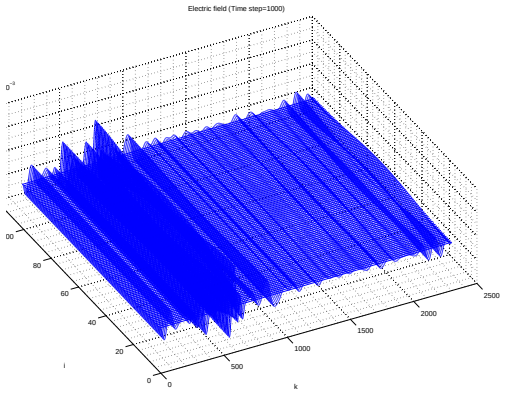


Fig. 2: Dynamic range of the electric field values

2. Hybrid floating-point computation

Figure 1(a) shows the FDTD algorithm [1]. It starts with an initial data of electric and magnetic fields. The initial data are processed to obtain the electric field for the first time step. After that, the boundary conditions are applied. Then the magnetic field data are obtained and the boundary conditions are applied. This process continues for n time steps. Figure 1(b) shows the electric and magnetic field computations. Electric and magnetic fields (in x, y, z directions) are denoted by E and H respectively. The coordinates of the 2D fields are denoted by (i, k).

To increase the speed-up, we observe the characteristics of our application [3]. According to [3], the electromagnetic field outside the cavity is weaker than that inside. Figure 2 shows the computation grid for FDTD calculation. The electric field far away from the grid origin has a small dynamic range. We observed similar characteristics from

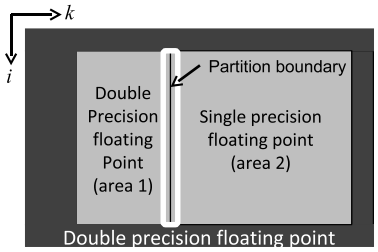


Fig. 3: Partition for single/double precision computation

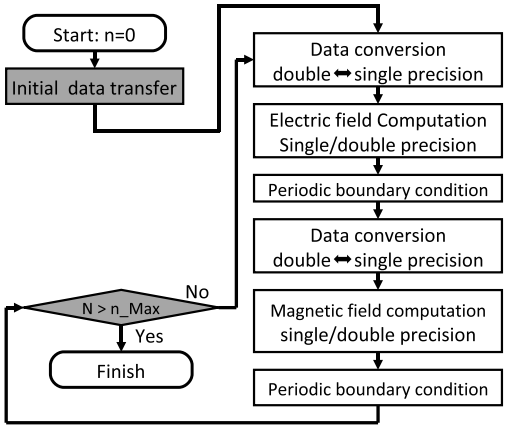


Fig. 4: Hybrid single/double precision computation

the magnetic field data analysis also. Therefore, we use double-precision floating-point when the dynamic range is large and single-precision floating-point when the dynamic range is small. As shown in Fig.3, the entire boundary and "area 1" are done in double precision floating-point while the rest of the computations in "area 2" are done in single precision floating-point. One problem in this method is the float-to-double conversion overhead. As shown in Fig.1(b), calculation of electric field of the coordinate (i, k) requires the magnetic fields at its left and right coordinates. Similarly, calculation of magnetic field of coordinate $(i, k + 1/2)$ requires the electric field at its left and right coordinates. To compute the electric and magnetic fields on the partition boundary in Fig.3, we need both double and single precision values that belong to area 1 and area 2 respectively. Therefore, we need float-to-double conversion and this is an additional overhead. Figure 4 shows the flow-chart of the hybrid single/double precision floating-point computation.

3. Evaluation

For the evaluation, we use Intel core i7 960 CPU and GeForce GTX 590 GPU. As shown in Fig.5, the proposed method is 1.79 times faster than the conventional double-precision GPU implementation [5]. Double-precision floating-point computation requires two CUDA cores while single-precision requires only one. This is the reason for the speed-up of the proposed method.

Figure 6 shows the processing time against the amount of double precision computation. Note that, the "double precision computation area" refers to the percentage of the

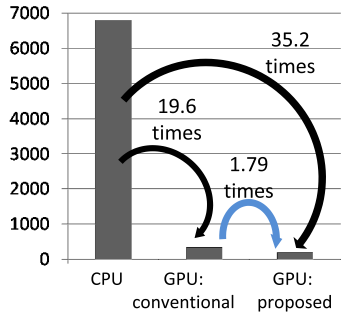


Fig. 5: Processing time comparison

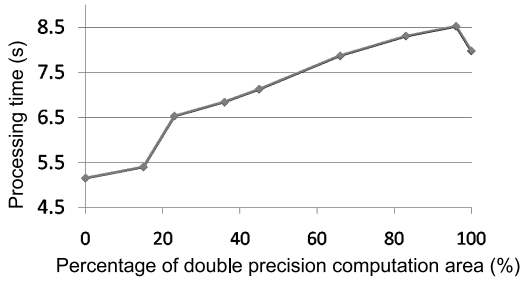


Fig. 6: Accuracy of the hybrid computation

grid area done in double-precision. The processing time increases with the amount of double precision computation. However, the processing time of "95% double precision computation" is larger than that of "100% double precision computation". This is due to the float to double conversion overhead in hybrid single/double precision computation.

4. Conclusion

In this paper, we proposed a hybrid single/double precision floating-point computation to accelerate FDTD on GPU. Since the amount of hardware required for double-precision is two times larger than that of the single-precision, we can increase the performance of the GPU by doing more computation in single-precision. However, due to the double-to-float conversion overhead, at least 85% of single-precision computation is needed for a considerable speed-up.

References

[1] H. S. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media", IEEE Transactions on Antennas and Propagation, Vol.14, No.3, pp.302-307, 1966.

[2] T. S. Ibrahim,R. Lee, B. A. Baertlein, Y.Yu, and P.M. L.Robitaille, "Computational analysis of the high pass birdcage resonator: finite-difference time-domain simulations for high-field MRI", Magn. Reson. Imag., vol.18, no.7, pp.835-843, 2000.

[3] Y. Ohtera, S. Iijima and H. Yamada, "Cylindrical Resonator Utilizing a Curved Resonant Grating as a Cavity Wall", micromachines, pp.101-113, 2012.

[4] A. Taflove and M. E. Brodwin, "Numerical Solution of Steady-State Electromagnetic Scattering Problems Using the Time-Dependent Maxwell's Equations", IEEE Transactions on Microwave Theory and Techniques, Vol23. Issue 8, pp.623-630, 1975.

[5] Z. Bo, X. Zheng-hui, R. Wu, L. Wei-ming, S. Xin-qing, "Accelerating FDTD algorithm using GPU computing", International Conference on Microwave Technology & Computational Electromagnetics (ICMTCE), pp.410-413, 2011.