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.

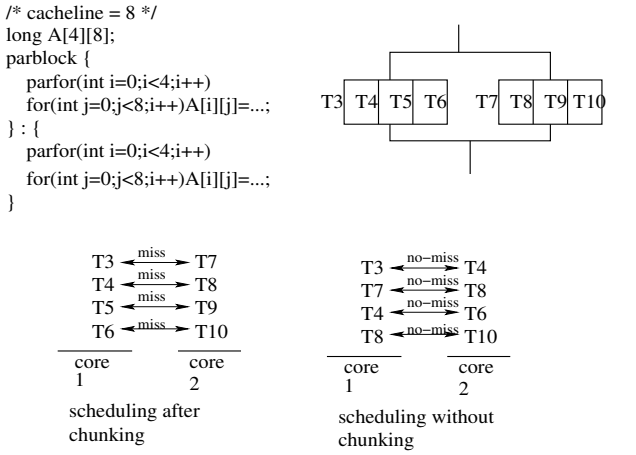

Fig. 1: Lazy threads prevent cache aware scheduling

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.

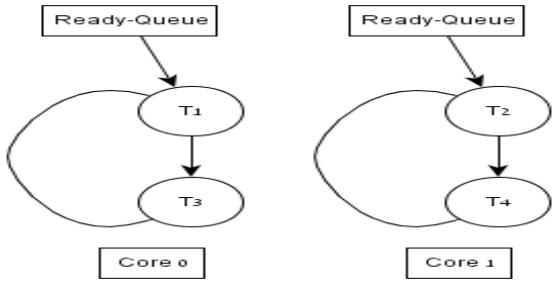

Fig. 2: Executing a program

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}.

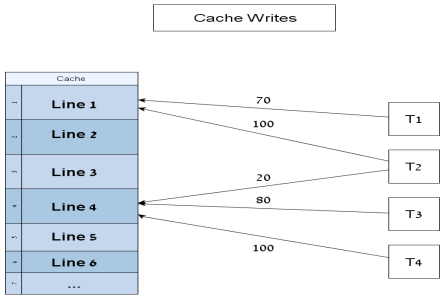

Fig. 3: Cache statistics obtained for the execution of figure 2.

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.

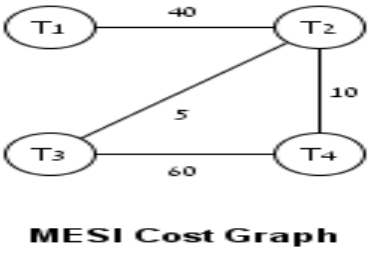

Fig. 4: MESI statistics obtained for the execution of figure 2.

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

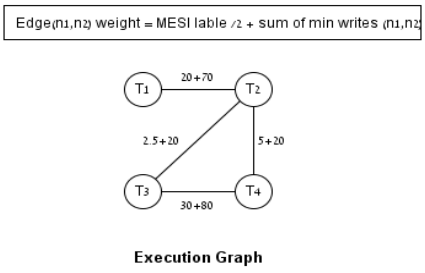

Fig. 5: Unifying MESI profile and the cache profile for the execution example of figure 2. $edge(T_{in}, T_{im}) = 0.5 * MESI(T_{in}, T_{im}) + \sum_{j=1}^{cache\ lines} minimum\ cache - profile(T_{in,j}, T_{im,j})$

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.

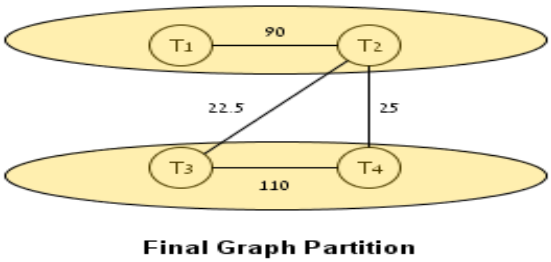

Fig. 6: Graph partition of the unified profile graph of figure 5.

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.

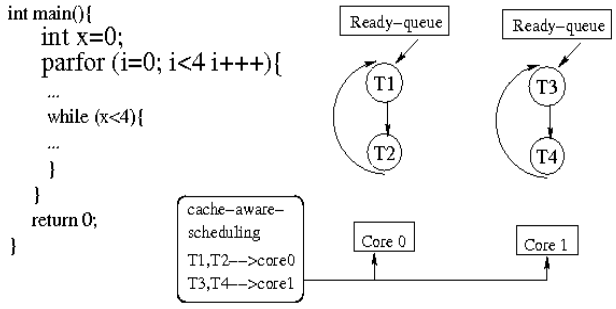

Fig. 7: New execution guided by the graph partition of figure 6.

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.